ATP C.N.R.S. 4273
Intelligence Artificielle

Amélioration, Compréhension, Accélération automatiques de programmes LISP

Responsable: Patrick Greussay

Rapport intermédiaire

par

J. Chailloux
D. Goossens
P. Greussay
H. Wertz

Octobre 1980

C.N.R.S. LA 248, L.I.T.P.
2 Place Jussieu
75005 Paris

Université Paris-8-Vincennes
2 Rue de la Liberté
Saint Denis, 93200

Présentation

Ce rapport intermédiaire rend compte des activités du groupe d'Intelligence Artificielle de l'Université Paris-8-Vincennes rattaché au LA 248 dans le cadre de l'ATP 4273. Ces activités sont essentiellement consacrées aux implémentations et améliorations ainsi qu'à la diffusion du langage VLISP et à ses applications principales parmi lesquelles la compréhension automatique de programmes et l'aide à la programmation.

En 1980, une nouvelle application a été développée par le groupe: les éditeurs vidéo couplés à VLISP, et en particulier les systèmes de fenêtrage dynamiques et interactifs. Ces applications sont en cours de développement pour les systèmes VLISP 8 et VLISP 11, et un nouvel éditeur vidéo interactif pleine page: TTV, a été implémenté sur une large variété d'ordinateurs de type PDP-11 (11/40, 11/34, 11/55, LSI-11, Plessey, Heathkit) et a été rendu disponible pour les terminaux VT50, VT52, Perkin Elmer 550, VT100, H88, Televideo Inc, VT11 graphique. Cet éditeur est en cours de diffusion et est utilisé à l'Université Paris-8, à l'IRCAM, à l'Université de Stanford, à l'université Paris-Sud ainsi qu'à l'Université de Poitiers.

Le présent rapport intermédiaire présente les travaux du groupe dans ces domaines, et inclut pour chacun d'entre eux une large bibliographie.

Sommaire

Le modèle VLISP : description, implémentation et évaluation

Jérôme CHAILLOUX

Octobre 1980

Notre étude présente la réalisation de ***trois systèmes*** VLISP (dialecte du langage LISP) développés à l'université de Paris 8 - Vincennes. Ces systèmes ont été réalisés :
- sur micro-processeur à mots de 8 bits (Intel8080/Zilog80)
- sur mini-ordinateur à mots de 16 bits (PDP11)
- sur gros ordinateur à mots de 36 bits (PDP10)

De ces réalisations est extrait un ***modèle d'implémentation***.

Cette étude propose des solutions aux problèmes de construction et d'évaluation de tels systèmes. Ces problèmes sont :

1) La description exhaustive des implémentations.
Nous proposons une description fondée sur la ***machine référentielle*** VCMC2.

2) La représentation adéquate des objets et des fonctions VLISP.
Nous avons associé des ***propriétés naturelles*** aux objets dès leur création et nous avons établi une ***typologie fonctionnelle*** de ces objets.

3) L'efficacité de l'interprète (en place, en temps d'exécution et en compréhension). Notre interprète effectue, pour ses besoins propres, une allocation optimale de la mémoire (allocation mesurée en terme d'appels du module CONS). L'accès direct (ne nécessitant qu'un accès mémoire) aux valeurs des objets de type variable et fonction, et la classification des fonctions par types permettent un ***lancement immédiat*** de toutes les fonctions. La transparence de nos méthodes de description est une conséquence naturelle des deux choix précédents.

4) Le ***pouvoir*** des structures de contrôle.
Notre modèle d'implémentation généralise les structures de contrôle de VLISP **SELF** et **ESCAPE** en intégrant les nouvelles constructions **EXIT**, **WHERE** **et** **LETF** et en unifiant totalement leur description et leur implémentation.

5) La souplesse d'utilisation du système.
Nous introduisons le nouveau concept de ***CHRONOLOGIE*** qui permet de créer dynamiquement de nouveaux évaluateurs, permettant les traces méta-circulaires.

Une incarnation de notre modèle est donnée dans [CHAILLOUX 80], sous la forme de la réalisation d'un système VLISP pour la machine référentielle VCMC2.

1.0 HISTORIQUE CRITIQUE DE L'IMPLEMENTATION DE LISP.

Le langage LISP [BERKELEY 74, WEISSMAN 67, ALLEN 78] est aujourd'hui le langage le plus utilisé dans les domaines de l'Intelligence Artificielle [WINSTON 77] et de la théorie de la programmation [NIVAT 79]. La particularité majeure de LISP est d'avoir pu évoluer de façon naturelle depuis sa naissance en 1960 [McCARTHY 60a, 60b], à mesure qu'apparaissaient de nouveaux besoins. L'échec patent des rares tentatives de standardisations *{Note 1}* a d'une part permis sa survie et d'autre part conduit à une amélioration constante du langage tant en efficacité à l'interprétation qu'en pouvoir de ses concepts.

Il en résulte aujourd'hui une floraison d'implémentations, bien loin du premier LISP 1 [McCARTHY 60a], ayant chacune des spécifications particulières en fonction de leur utilisation et du moment de leur création.

On distingue ainsi 4 grandes familles d'implémentations :

1) La lignée LISP 1.5 [McCARTHY 62]. Descendante directe de LISP 1 qui était un système interactif (avec le "FLEXOWRITER system"), elle a donné le LISP 1.6 [QUAM 72] très proche de LISP 1.5, mais ayant abandonné la liaison par A-liste, décision dont les conséquences ont été monumentales. Par la suite, l'importance capitale de l'intégration des instruments de mise au point devint de plus en plus évidente, et LISP 1.6 fut augmenté d'un éditeur et d'aides à la mise au point pour donner le U.C.I. LISP [BOBROW 73a].

2) Les systèmes INTERLISP (anciennement BBN LISP) [TEITELMAN 75] orientés sur une utilisation exclusivement interactive, et qui ont donnés naissance à des sous-systèmes tels que CLISP [TEITELMAN 73] ou DLISP [TEITELMAN 77] qui font une utilisation intensive de périphériques interactifs tels les écrans à haute résolution.

3) MACLISP [MOON 74, LAUBSCH 76] au M.I.T. dont le développement s'est réalisé en symbiose avec celui du système MACSYMA [MACSYMA 75] et qui a inspiré la conception de la machine LISP du M.I.T. [WEINREB 79].

4) Enfin les systèmes VLISP [GREUSSAY 76a, CHAILLOUX 78a, CHAILLOUX 80] développés en France à l'Université de Paris 8 - Vincennes, implantés sur les 3 catégories de systèmes matériels (gros, mini et micro) et qui sont dotés des structures de contrôle les plus puissantes.

Le point commun de ces familles d'implémentations est d'avoir toutes une réalisation sur l'ordinateur de référence PDP-10 [DEC 78a], qui est la machine la plus utilisée dans le cadre des recherches en Intelligence Artificielle. Pour cette raison les performances de ces différentes réalisations sont aisément mesurables.

En plus du PDP-10, ces systèmes LISP n'ont été implantés que sur un nombre restreint d'autres matériels en général de taille très importante tels l'IBM série 360/370 [BOLCE 68, HAFNER 74], l'UNIVAC 1100 ou l'IRIS-80 pour lequel on dénombre 3 réalisations en France, le TLISP à l'Université Paul Sabatier de TOULOUSE [DURIEUX 78], le RLISP au laboratoire IMAG de GRENOBLE [LUX 78]

{Note 1} ces standards sont aussi prématurés qu'arbitraires. Prématurés car ils tombent en désuétude aussitôt créés (voir le standard de [HEARN 69] puis [MARTI 79]) et ne tiennent pas compte de l'évolution du langage, arbitraires car ils sont essentiellement définis pour satisfaire un programme particulier (REDUCE 2 dans le cas de HEARN [HEARN 73, 74]).

et SIRLISP à l'E.N.S.T de Paris [COILLAND 79].

LISP 1.5 et LISP 1.6, hormis des réalisations-jouets à fins pédagogiques, sont à présent tombés en désuétude.

MACLISP et INTERLISP, trop dépendant de la machine PDP10 (et même du système d'exploitation utilisé respectivement ITS et TENEX) n'ont pu être transportés dans leur intégralité sur d'autres machines *{Note 1}*.

Seul, VLISP, (du fait probablement de sa conception en Europe et de la diversité des ordinateurs qui y sont disponibles) a pu (ou a dû) être réalisé sur un grand nombre de machines très variées telles que CAE510 [GREUSSAY 72], CAB500 [WERTZ 74], T1600 [GREUSSAY 75], PDP10 [CHAILLOUX 78c], SOLAR16 [GREUSSAY 78b], 8080 et Z80 [CHAILLOUX 79a] et PDP11 [GREUSSAY 79b].

La diversité de toutes ces réalisations a entrainé la naissance d'histoires (et d'historiens) de LISP, apportant de précieuses informations sur l'évolution naturelle des langages de programmation *{Note 2}* [McCARTHY 78], [WHITE 78], [STOYAN 78a,78b].

Les recherches actuelles de construction des systèmes LISP s'élaborent pour l'essentiel selon les 4 axes suivants :

- implémentation sur des ordinateurs universels classiques tel le VAX-11 [DEC 78e] ou le S-1 [HAILPERN 79].
- implémentation sur des ordinateurs spécialisés, voire des micro-ordinateurs, à ressources limitées [CHAILLOUX 78a, TAFT 79].
- réalisation de machines spécialisées [GREENBLATT 74, KNIGHT 74, SHIMADA 76, LISPMACHINE 77, LECOUFFE 77, TAKI 79, WEINREB 79].
- apparition d'unités centrales, en technologie V.L.S.I. [MEAD 80], capables d'interpréter directement LISP à une translation près du langage source en sa représentation arborescente, dont le répertoire des sommets constitue un jeu d'instructions d'interprétation directe de LISP (les CHIP-LISP) [STEELE 79, HOLLOWAY 80].

{Note 1} En date de Septembre 1978, MACLISP ne tournait encore qu'avec beaucoup de difficultés, en ce qui concernait ses aspects-système évolués, au laboratoire d'Intelligence Artificielle de Stanford, sous système WAITS [GREUSSAY 78a].

{Note 2} Par opposition aux cas de langages à génération spontanée ou à comités [HORNING 79], à définition discrétionnaire.

2.0 COMMENT DECRIRE LES IMPLEMENTATIONS : la machine VCMC2.

Dès son apparition LISP a été exposé méta-circulairement *{Note 1}* afin de décrire ses propres implémentations.
Cette utilisation de LISP, très commode, ne satisfaisait pas les implémenteurs experts, qui n'avaient à leur disposition que ces descriptions méta-circulaires accompagnées, dans le meilleur des cas, du texte d'une implémentation particulière. Tous les problèmes réels d'implémentation en machine étaient soit totalement ignorés (dans le cas où LISP était utilisé à sa propre description) soit entièrement orientés vers une machine particulière.

Pour illustrer cette situation, nous utiliserons dans cette section, une succession progressive de descriptions du module classique EVLIS : nous partirons de sa description méta-circulaire en LISP, nous la qualifierons sur des machines particulières, enfin nous en donnerons la description dans notre modèle de référence.

Voici la description de la fonction EVLIS, en LISP classique, telle quelle est donnée dans les manuels de référence :

LISP classique

```
(DE EVLIS (L)
   (COND
     ((NULL L) NIL)
     (T (CONS (EVAL (CAR L)) (EVLIS (CDR L))))))
```

Une telle description purement méta-circulaire était déjà remise en question par John McCARTHY, le créateur du langage : il introduisit deux langages LISP, un langage algorithmique (le Méta-langage) qui utilisait des M-expressions et un langage de programmation (le LISP tel qu'on le parle encore) qui mettait en jeu les S-expressions [McCARTHY 62]. Ce double langage permettait de distinguer les données décrites en S-expression des programmes décrits sous forme de M-expressions [PERROT 79].

{Note 1} LISP est spécifiable par un interprète dit méta-circulaire : un tel interprète, écrit en VLISP, ramasse en une seule description la spécification du langage et celle de l'interprète lui-même (voir la description de l'évaluateur VLISP-10 en VLISP-10 donné dans [CHAILLOUX 78c] page 20).

La traduction du langage algorithmique vers le langage de programmation LISP devait s'effectuer automatiquement *{Note 1}*. on sait aujourd'hui que seul le langage de programmation a survécu.

Voici donc la description de la fonction EVLIS en utilisant la notation sous forme de M-expressions :

<u>M-expressions</u>

```
evlis[l]=[null[l]→NIL;T→cons[eval[car[l]];evlis[cdr[l]]]]
```

Si on considère qu'une implémentation particulière constitue une description de référence on se heurte aux deux difficultés que sont :

1) l'opacité inhérente à la méconnaissance préliminaire de l'ordinateur source

2) la non-généralité chronique de ce type de description.

Voici donc cette même fonction telle qu'elle est codée dans le langage machine du PDP10 [CHAILLOUX 78c] :

<u>Langage machine PDP-10</u>

```
EVLIS:
    JUMPE   A1,VPOPJ        ; pas d'argument
    HRRZ    A2,MEM(A1)      ; A2 ← (CDR A1)
    PUSH    P,A2            ; qui est sauvé
    PUSHJ   P,EVALCA        ; évalue la 1ère val.
    HLRZ    A1,MEM(A1)      ; A1 ← (CONS A1)
    EXCH    A1,MEM(FREE)
    EXCH    FREE,A1
    POP     P,A2            ; récupère le reste
    CAMGE   A2,BLIST        ; au moins 2 ?
    JRST    VPOPJ           ; non : fini
    PUSH    P,A1            ; sauve le 1er doublet
    PUSH    P,A1            ; sauve le dernier
    PUSH    P,A2            ; sauve le reste
    MOVEI   A1,(A2)         ; A1 ← les arguments
EVLIS1:
    HRRZ    A2,MEM(A1)      ; A2 ← (CDR A1)
    MOVEM   A2,(P)          ; dans le sommet de la pile
    PUSHJ   P,EVALCA        ; évalue l'argument suivant
    HLRZ    A1,MEM(A1)      ; A1 ← (CONS A1)
```

{Note 1} c'est l'absence des caractères spéciaux [,], et → sur la perforatrice de cartes IBM026 qui a empéché l'utilisation systématique des M-expressions.

```
21        EXCH     A1,MEM(FREE)
22        EXCH     FREE,A1
23        MOVE     A2,-1(P)          ; A2 ← le dernier doublet
24        HRRM     A1,MEM(A2)        ; (RPLACD A2 A1)
25        MOVEM    A1,-1(P)          ; sauve le dernier doublet
26        MOVE     A1,(P)            ; A1 ← le reste
27        CAML     A1,BLIST          ; il reste des éléments ?
28        JRST     EVLIS1            ; oui.
29        SUB      P,[3,,3]          ; non : nettoie la pile .
30        MOVE     A1,1(P)           ; A1 ← 1er doublet
31   VPOPJ:
32        POPJ     P,                ; et voilà.
```

Le point 1) est illustré ici par l'instruction MOVEI, utilisée à la ligne **15**, qui emploie classiquement la propriété d'adressage suivante :

immédiat + indexé = direct

et permet un gain substanciel de temps :

Temps d'exécution des instructions		
MOVE	A1,A2	1.14 micro-sec
MOVEI	A1,(A2)	0.62 micro-sec

sur l'unité centrale PDP 10 KI [DEC 78a].

On notera également la confusion conceptuelle introduite par l'utilisation du pointeur de pile P comme registre d'index, pointeur qui ira même (voir la ligne 30) jusqu'à être employé à l'extérieur de la zone pile.

La non-généralisation de cette description est malheureusement une conséquence de l'utilisation raisonnablement experte d'une machine particulière.

Nous devons aller au delà de ces 3 représentations pour aborder, décrire correctement, résoudre enfin les véritables problèmes d'implémentation en machine :

- la représentation en LISP classique montre assez bien l'intention de la fonction mais ignore complètement les problèmes d'allocation des ressources, souvent limitées, ainsi que la gestion de la récursion.

- la représentation en M-expressions tout en évoquant approximativement une notation algébrique présente les mêmes inconvénients que la représentation en LISP et introduit un niveau supplémentaire de traduction.

- la représentation en langage machine PDP10 fait surgir de nouveaux problèmes locaux tels que l'accès aux demi-mots de la machine voire l'utilisation du pointeur de pile comme registre d'index. La description en langage machine est alors supplantée par l'expérience nécessaire à la mise en jeu correcte d'adressages et de jeux d'instructions particuliers.

Nous proposons donc de représenter l'implémentation au moyen de programmes d'une machine de référence, la machine VCMC2, descendante directe de la machine VCMC1 [CHAILLOUX 78b], et indépendante des représentations internes des objets eux-mêmes. C'est de cette description que naissent naturellement les incarnations opérationnelles particulières de VLISP sur les ordinateurs les plus diversifiés. Cette machine, décrite dans [CHAILLOUX 80], est simulée en VLISP-10 sur ordinateur PDP-10.

La description de VLISP très concise, qui en découle dégage les caractéristiques fondamentales de l'implémentation de modules tels que EVLIS en machine, i.e. :

1) les actions de base (test par rapport à NIL, CDR, CONS ...)

2) la gestion de la récursion par utilisation d'une pile et d'un accumulateur.

3) le mécanisme du sauvetage et de la restauration du reste de la liste et du résultat intermédiaire au travers de la pile.

4) l'internalisation totale des fonctions interfaces : entrées/sorties et traces (ce que ne pouvait pas faire la notation FILTRE de [GREUSSAY 76b]).

Voici donc la description, en VCMC2, de l'implémentation en machine de la fonction EVLIS :

La fonction EVLIS décrite en VCMC2

```
EVLIS:  TNIL    A1,,[RETURN]
        CDR     A1,TST,[CALL (EVCAR)]
        XTOPST  A1,,[CALL (EVLIS)]
        CONS    TST,A1,[RETURN]
```

Cette représentation générale atteint le degré de précision suffisant pour :

1) appréhender la véritable structure en actions de base de l'implémentation (ce que ne pouvait pas faire la description d'INTERLISP *{Note 1}*

{Note 1} la notation utilisée dans la description de [MOORE 76] ressemble à la notation sous forme de M-expressions et ne rend pas compte de la gestion de la récursion et des résultats intermédiaires. Sachant que EVLIS est la version SUBR du module LIST (de type FSUBR) [MOORE 76 pp. 11] ne peut décrire cette dernière qu'elliptiquement par le dispositif typographique des points de suspension :

LIST[x1;x2;...xk]
Return CONS[x1;CONS[x2;...CONS[xk;NIL]...]]

2) expliciter totalement la gestion de la pile de récursion en unifiant dans l'adressage la mise en jeu des accumulateurs et du cache de pile.

3) indiquer la circulation dans la pile des états d'évaluation intermédiaires du contenu initial du registre A1

4) préciser enfin la structure de contrôle par attachement à chaque instruction d'une continuation complexe.

3.0 EFFICACITE ET PUISSANCE DE L'INTERPRETATION.

Du fait de sa représentation des objets et des fonctions, notre modèle permet de réduire considérablement le temps d'interprétation et de disposer d'un grand nombre de structures nouvelles, qui facilitent l'écriture des programmes, en améliorent la lisibilité et en diminuent sensiblement la taille.

Voici la définition de la fonction de FIBONACCI utilisée pour des comparaisons de vitesse sur plusieurs interprètes LISP fonctionnant sur PDP10 :

```
(DE FIB (n)
    (COND
      ((ZEROP n) 1)
      ((EQ n 1) 1)
      (T (PLUS (FIB (SUB1 n)) (FIB (DIFFER n 2))))))

(FIB 20) ☞ 10946
```

Voici les temps relevés sur MACLISP, INTERLISP, VLISP-10 et VLISP-11 *{Note 1}* pour le calcul de l'appel **(FIB 20)** :

[Temps en secondes]			
MACLISP	INTERLISP	VLISP-10	VLISP-11
48.66	105.78	13.34	35.2

La rapidité remarquable de l'interprétation de notre modèle est dûe :

1) à l'utilisation intensive des propriétés naturelles des atomes et principalement du type associé à chaque fonction. Le type d'une fonction LISP est déterminé par le langage dans lequel elle est écrite, le mode de passage de ses arguments (les arguments sont ou ne sont pas évalués) ainsi que le nombre de ses arguments. Cette typologie est abondamment décrite dans [CHAILLOUX 80 chapitre 4]. Ce type est directement accessible, ce qui permet de réaliser en même temps et le test du type et l'appel de la routine associée, au moyen d'un seul branchement indirect indexé. En terme de syntaxe, VLISP utilise donc l'appel par nom des

{Note 1} MACLISP utilise dans le cadre de ce test une unité centrale PDP10 KA-10 et des mémoires à 1.8 micro-secondes [WHITE 76], INTERLISP une unité centrale PDP10 KA-10 et des mémoires à 1.0 micro-secondes, VLISP-10 une unité centrale PDP10 KI-10 et des mémoires à 1.0 micro-secondes. VLISP-11 n'a à sa disposition qu'une micro-unité centrale LSI 11/02.

fonctions.

2) au fait que l'interprète ne réalise plus aucun CONS pour ses besoins propres. L'utilisation exclusive d'une pile commune de données et de contrôle évite de créer des structures intermédiaires (sous forme de cellules de liste) durant le processus d'évaluation. Le nombre de récupérations de la mémoire dynamique se trouve ainsi réduit aux seuls besoins de l'utilisateur *{Note 1}*.

3) à l'interprétation itérative des fonctions récursives de type NPR et CPR [GREUSSAY 76c].

4) à l'utilisation des structures de contrôle de VLISP qui permettent une écriture plus concise et une interprétation plus rapide. En particulier :

   - l'opérateur point-fixe du λ-calcul [ROBINET 78], représenté par la fonction **SELF** de [GREUSSAY 77]

   FIBONACCI se reécrit alors commodément :

```
(DE FIB (n)
    (IF (ZEROP n) 1
        (IF (EQ n 1) 1
            (PLUS (SELF (SUB1 n))
                  (SELF (DIFFER n 2))))))
```

   - les fonctions d'échappement **ESCAPE**. Introduites par [GREUSSAY 76b], elles permettent d'associer dynamiquement des fonctions d'échappement à des atomes, réalisant ainsi des sorties non-locales.

5) enfin à de nouvelles structures de contrôle. Ces nouvelles structures de contrôle vont pouvoir réaliser :

   - la description de fonctions anonymes (i.e. qui ne sont pas associées à des noms) en utilisant la forme **INTERNAL**.

   - les sorties locales à une fonction en utilisant la fonction **EXIT**.

   - la redéfinition dynamique de fonctions de n'importe quel type en utilisant la primitive **WHERE**.

   - la définition d'une nouvelle classe de fonctions, les variables fonctions, apportant une solution nouvelle au problème des variables internes.

{Note 1} La gestion de la mémoire en LISP est automatique et dynamique ce qui assure une utilisation optimum de la place mémoire mais oblige à construire des modules d'accès et d'allocation (le module CONS) spécialisés ainsi qu'un récupérateur de mémoire, dispositif qui ralentit l'interprète.

3.1 Description de fonctions anonymes

Notre modèle permet de décrire des fonctions anonymes (non associées à des symboles atomiques) de n'importe quel type au moyen d'une des deux formes **INTERNAL** suivantes :

```
(INTERNAL type adresse-mémoire)
(INTERNAL type fonction)
```

La première forme permet de décrire des fonctions écrites en langage machine, et la seconde des fonctions écrites en VLISP.

Ces deux nouvelles formes généralisent l'ancienne utilisation de la forme LAMBDA (qui a été gardée par souci de compatibilité et de simplification de l'écriture des EXPR anonymes) pour tous les types de fonctions et permettent ainsi de limiter l'utilisation des noms associés aux fonctions.

Cette limitation des noms réduit d'autant plus l'espace mémoire nécessaire au stockage des symboles atomiques et accroit la lisibilité du programme qui ne contient plus que les noms indispensables.

3.2 Sorties locales

La forme **EXIT** permet de sortir de la dernière fonction invoquée de type EXPR, FEXPR ou MACRO. La valeur retournée est évaluée dans l'environnement précédant exactement le retour de cette fonction donc évaluée en position terminale. Cette propriété est utilisée pour forcer un traitement de récursion terminale *{Note 1}* ce qui n'était pas le cas des fonctions **RETURN** des anciennes formes **PROG** *{Note 2}*.

Voici la méta-description de la fonction **MEMBER** qui utilise la structure de contrôle itérative **WHILE**.

{Note 1} Ces appels récursifs terminaux sont interprétés d'une manière itérative beaucoup plus efficiente.

{Note 2} La forme **PROG** *a été abandonnée dans notre modèle (ainsi que ses fonctions associées* **GO**, **GOTO** *et* **RETURN***) car les nouvelles fonctions de sorties locales et non-locales sont à la fois plus rapides (en effet le balayage préliminaire de tous les corps de* **PROG** *pour construire la table des étiquettes a disparu) et plus générales (avec la possibilité de retourner directement d'un nombre quelconque d'appels).*

```
(DE MEMBER (a l)
   (WHILE L
      (IF (EQUAL (CAR l) a)
         (EXIT l)
         (NEXTL l))))
```

L'appel de la fonction **EXIT** permet une sortie extraordinaire du **WHILE**.

3.3 Définitions dynamiques de fonctions

La forme WHERE permet de définir dynamiquement de nouvelles fonctions et de rendre fluides *{Note 1}* les fonctions elles-même.

Voici la syntaxe de cette nouvelle forme :

```
(WHERE (<nom> <fval>)  <s1> ... <sN>)
```

Le premier argument du **WHERE** est une définition temporaire d'une fonction **<fval>** associée au nom **<nom>**. Le reste des arguments du WHERE i.e. **<s1>** ... **<sN>** est un corps dans lequel la définition précédente est active. Au sortir du **WHERE**, la définition associée au nom **<nom>** disparait et sa définition antérieure (s'il en possédait une) est restaurée.

Voici la fonction de traçage d'une courbe Dragon [GARDNER 67, SCHOETTL 75] en LOGO-LISP [WERTZ 79] dans laquelle la fonction **tourne** est redéfinie dynamiquement.
Les fonctions **<avance>**, **<droite> et <gauche>** sont des primitives LOGO :

```
(DE tourne () (<droite> 90))

(DE dragon (n l)
   (IF (ZEROP n)
      (<avance> l)
      (WHERE (tourne '(() (<gauche> 90)))
         (dragon (SUB1 n) l))
      (tourne)
      (WHERE (tourne '(() (<droite> 90)))
         (dragon (SUB1 n) l))))
```

{Note 1} Une variable fluide est une variable libre liée dynamiquement.

3.4 Les variables fonctions

Notre modèle propose une solution au problème de l'accès aux variables internes *{Note 1}* de l'interprète. En effet, un choix crucial d'implémentation se pose.
Il n'est que trop tentant d'utiliser des variables VLISP pour accèder à ces variables internes. Appelons OBASE la variable interne qui contient à tout moment la base de numération des nombres en sortie. Une affectation incorrecte de cette variable (par exemple avec une valeur négative), détruirait toute possibilité future d'impression de nombres.
Une deuxième stratégie consiste à utiliser une fonction associée à chaque variable interne. Cette fonction va contrôler la validité des valeurs affectées à la variable. C'est ce qu'on entend par variabilisation ou fonctionnalisation de l'accès.

C'est cette dernière stratégie qui a été adoptée dans notre modèle. De plus celui-ci propose une nouvelle classe de fonctions, les variables fonctions, qui permettent dans une stratégie de fonctionnalisation des variables internes de lier dynamiquement les valeurs de ces variables internes, retrouvant ainsi toutes les propriétés des variables.
En lecture ces variables fonctions n'ont pas d'argument et en écriture elles possèdent un argument (évalué) qui est la nouvelle valeur de la variable fonction.

```
(variable-fonction)          :  lecture
(variable-fonction  valeur)  :  écriture
```

La nouvelle structure **LETF** réalise la liaison dynamique des variables fonctions. Elle possède une syntaxe identique à la macro **LET** :

```
(LETF (variable-fonction  valeur) corps-du-LETF)
```

{Note 1} rappelons que ces variables internes sont les mots mémoires de travail de l'interprète et qu'il est souhaitable que l'utilisateur ait accès à certaines de ces variables pour faciliter l'utilisation du système et en augmenter la puissance.

Cette forme est exécutée en 4 temps :

1) appel de la variable fonction en lecture (sans argument). La valeur retournée est conservée.

2) appel en écriture de la variable fonction avec l'argument transmis au **LETF**.

3) exécution en séquence du corps du **LETF** qui calcule la valeur retournée par la forme **LETF**.

4) re-appel en écriture de la variable fonction avec en argument son ancienne valeur conservée en 1).

Ce type de liaison (de même que celle des variables) est réalisée automatiquement et en particulier le point 4) est effectuée en cas de sortie extraordinaire par un **EXIT** ou un **ESCAPE** enveloppant.

L'utilisation de ces variables fonctions permet donc d'associer à des variables des fonctions qui réalisent des contrôles d'accès dynamiques tant en lecture qu'en écriture.

Voici un exemple d'utilisation de la variable fonction **OBASE** qui contrôle la base de conversion des nombres en sortie. La fonction **PRINT-base-x** imprime son 2ème argument **n** dans la base de numération **x** fournie en 1er argument :

```
(OBASE)    ☞    10

(DE PRINT-base-x (x n)
    (LETF (OBASE x)
       (PRINT n)))

(PRINT-base-x 16 1000)   ☞   3E8

(OBASE)    ☞    10
```

4.0 LA SOUPLESSE D'UTILISATION.

Un certain type de programmation-système consiste à implanter, valider (ou invalider), mesurer, modéliser de nouveaux types d'architectures logicielles ou matérielles (par simulateur), à la limite de l'élaboration de modèles en Intelligence Artificielle.
Cette programmation se fera tout naturellement dans les conditions les plus interactives et les plus expérimentales possibles, et nécessitera des outils permettant la réalisation la plus rapide du modèle à valider. Cette activité demande donc un langage puissant, très facilement et très rapidement mis en oeuvre, car les temps de vie des différents modèles sont extrêmement brefs et soumis à des modifications incessantes.

Ainsi notre système ne mésestime pas l'expertise de ses utilisateurs. Il s'agit d'un système utilisable et utilisé, illustrant la nécessité, parfois sous-estimée, de ne pas entraver l'expertise raisonnable de ses utilisateurs (par exemple en ne l'encombrant pas de contrôles statiques contraignant à une programmation médiocre) mais, tout au contraire, de leur fournir le maximum de pouvoir et de souplesse expressive.

L'utilisation réelle d'une tel système nécessite la faculté de contrôler par niveau le fonctionnement de l'interprète, ce qui amène tout naturellement à l'idée d'évaluateurs multiples selon les circonstances du calcul. Les outils de contrôle de chronologie, traces et erreurs qui vont être décrits en sont une illustration.

4.1 La notion de CHRONOLOGIE

Notre modèle dispose d'une multitude d'évaluateurs potentiels différenciés par un numéro d'ordre de création, un numéro de chronologie. L'accès à cette chronologie est réalisée au moyen de la variable fonction CHRONOLOGY.

Ces évaluateurs peuvent être appelés :

- par l'utilisateur au moyen de la fonction interprète EVAL dont le deuxième argument est le numéro de CHRONOLOGIE que l'on veut voir affecté à cette évaluation :

```
(EVAL expression chronologie)
```

- soit automatiquement par l'interprète, il s'agit alors d'interruptions. Ces interruptions sont déclenchées par des événements externes (horloges ou périphériques) ou par l'interprète lui-même dans les cas suivants :
  - appel de la boucle principale du système **(TOPLEVEL)**
  - erreur à l'interprétation **(ERROR)**
  - trace de l'évaluateur **(STEPEVAL)**
  - ligne pleine en sortie **(EOL)**
  - apparition d'une fin de fichier d'entrée **(EOF)**

Le traitement d'une interruption se fait par invocation d'une fonction propre à chaque type d'interruption, dans un nouvel évaluateur dont la chronologie est la chronologie précédente incrémentée d'une unité, i.e. :

```
(EVAL '(fonction associée à l'IT ....) (ADD1 (CHRONOLOGY)))
```

La valeur retournée par la fonction associée à l'interruption devient la valeur de l'interruption.

Il existe de plus une fonction de sortie extra-chronologie, la fonction **EXITCHRONOLOGY** qui réalise la sortie de la chronologie courante et retourne une valeur au créateur de cette chronologie. Cette fonction possède la même syntaxe que la fonction de sortie locale à une fonction, la fonction **EXIT**.

4.2 Les erreurs

Notre modèle ne provoque pas l'abandon du travail en cas de détection d'erreurs. Lorsqu'il se produit des états d'indétermination dans un évaluateur, celui-ci va dynamiquement créer un nouvel évaluateur. Sa tâche est de tenter de lever l'indétermination ou tout au moins de retourner une valeur à l'évaluateur interrompu.

Lorsque ces états d'indétermination sont détectés par un interprète (par exemple lors d'une consultation d'une variable non définie), une fonction spéciale, la fonction **ERROR**, est automatiquement invoquée avec comme arguments les objets indeterminés, dans un évaluateur différent (dont la chronologie est égale à la chronologie antérieure incrémentée d'une unité). La valeur ramenée par cette invocation est utilisée pour lever l'indétermination de l'interprète interrompu.

Cette fonction **ERROR** (comme toutes les fonctions d'interruptions) peut être redéfinie en VLISP afin d'utiliser des sytèmes complexes de corrections automatiques tel que le système PHENARETE de [WERTZ 78], des outils sophistiqués d'aides-à-la mise au point tel que le méta-évaluateur CAN de [GOOSSENS 77, 79], des analyseurs [WERTZ 77] des traces ou des steppers [GREUSSAY 79a]

Voici un exemple de la redéfinition de la fonction **ERROR** qui permet de construire une fonction testant si une forme peut être évaluée sans erreur par EVAL. Dans le cas ou EVAL produirait une erreur cette fonction retourne la valeur NIL, et dans le cas contraire elle retourne la valeur de l'évaluation. Cette dernière valeur est incluse en premier élément d'une liste pour pouvoir distinguer la valeur NIL correspondant à une évaluation de cette même valeur indiquant une erreur à l'évaluation.

```
(DE EVAL-teste-si-erreur (forme)
   (WHERE (ERROR '(() (erreur)))
     (ESCAPE erreur
        [(EVAL forme)])))
```

4.3 Les traces

Une des facilités majeures de notre modèle est d'avoir un mécanisme interne d'observation sélective de l'évaluateur lui-même en fonctionnement. Ce mécanisme permet d'appeler la fonction **STEPEVAL** (qui peut être bien entendu redéfinie en VLISP) avec comme argument la forme qui doit être évaluée, à chaque appel interne de l'évaluateur.

Ce mode trace est activé en donnant une valeur non-NIL au 3ème argument de la fonction interprète EVAL :

```
(EVAL expression chronologie trace)
```

Voici comment réaliser une trace de tous les appels de EVAL.

```
(DE TRACEVAL (exp)
   (WHERE
      (STEPEVAL (forme)
         (PRINT '—> forme)
         (SETQ IT (EVAL forme (SUB1 (CHRONOLOGY)) T))
         (PRINT '<— IT))
      (EVAL '(STEPEVAL exp) (ADD1 (CHRONOLOGY)))))

; Fonction de test : dernier élément de l ;

(DE FOO (l)
   (IF (NULL (CDR l)) (CAR l) (SELF (CDR l))))

; Appel de la trace ;

(TRACEVAL '(FOO '(A B)))

—>  (FOO '(A B))
—>  '(A B)
<—  (A B)
—>  (IF (NULL (CDR l)) (CAR l) (SELF (CDR l)))
—>  (NULL (CDR l))
—>  (CDR l)
—>  l
```

```
<—  (A B)
<—  (B)
<—  NIL
—>  (SELF (CDR l))
—>  (CDR l)
—>  l
<—  (A B)
<—  (B)
—>  (IF (NULL (CDR l)) (CAR l) (SELF (CDR l)))
—>  (NULL (CDR l))
—>  (CDR l)
—>  l
<—  (B)
<—  NIL
<—  T
—>  (CAR l)
—>  l
<—  (B)
<—  B
<—  B
<—  B
<—  B
<—  B

☞   B
```

Une des difficultés classiques de la réalisation du mécanisme de trace est posée par l'interférence entre la fonction traçante et la fonction tracée en particulier en cas d'utilisation des constructions **SELF** et **EXIT** : en effet dans la 11ème ligne de trace l'appel **(SELF (CDR l))** doit utiliser la fonction tracée (i.e. la fonction **FOO**) et non pas la dernière fonction appelée (i.e. la fonction **STEPEVAL** elle-même). De même l'évaluation d'une forme **EXIT** peut se rapporter soit au programme traçant soit au programme tracé. Pour lever toute ambiguité à l'évaluation des formes **SELF** et **EXIT**, notre modèle utilise la dernière fonction appelée dans la chronologie courante. La fonction traçante et la fonction tracée sont évaluées dans des chronologies différentes, ce mécanisme permettant également de réaliser des traces méta-circulaires *{Note 1}*.

Bien entendu des traces plus élaborées (comprenant des renfoncements correspondants aux niveaux d'imbrications des appels ou des numérotations de lignes par niveau et des possibilitées d'exécution incrémentales) peuvent être construites.

Le fait de pouvoir récupérer TOUS les appels internes de l'évaluateur permet, outre les traces, une modification complète de celui-ci.

{Note 1} KNUTH [KNUTH 69] propose page 211 l'exercice suivant :
"6. [40] Design a trace routine which is capable of tracing itself, in the sense of exercice 4; i.e., it should print out the steps of its own program at slower speed, and that program will be tracing itself at still slower speed, ad infinitum until memory capacity is exceeded."
Notre mécanisme de trace permet de ne tracer qu'un seul niveau de la fonction de trace.

Voici en exemple la fonction EVAL-NIL-si-UNDEF qui se comporte comme la fonction interprète standard EVAL mais qui ne provoque pas d'erreur à l'évaluation d'une variable indéfinie, se contentant uniquement de donner la valeur NIL à la variable et d'imprimer un message d'avertissement :

```
(DE EVAL-NIL-si-UNDEF (expression)
   (WHERE
      (STEPEVAL '((forme)
         (IF (OR (LISTP forme) (NUMBP forme) (BOUNDP forme))
             (EVAL forme () T)
             (PRINT "Je donne la valeur NIL à :" forme)
             (SET forme NIL))))
      (STEPEVAL expression)))
```

Voici quelques utilisations de cette fonction en supposant que les variables VARF001, VARF002 et VARF003 sont indéfinies

```
?   (EVAL-NIL-si-UNDEF 'VARF001)
        Je donne la valeur NIL à : VARF001
☞  NIL

?   (EVAL-NIL-si-UNDEF '['A VARF002 'B VARF003])
        Je donne la valeur NIL à : VARF002
        Je donne la valeur NIL à : VARF003
☞  (A NIL B NIL)
```

On notera dans cette fonction la redéfinition dynamique de la fonction STEPEVAL, qui permet de n'effectuer le test de variable indéfinie que dans la portée dynamique de la fonction EVAL-NIL-si-UNDEF.

5.0 LES SYSTEMES VLISP REALISES.

Une des caractéristiques les plus importantes de notre modèle est d'être très rapidement et très facilement implémenté sur les matériels actuels. Trois réalisations récentes montrent la diversité des machines pouvant recevoir ce système :

1) le système VLISP-10 [CHAILLOUX 78c], est réalisé sur l'ordinateur PDP KI 10 de Digital Equipment Corporation [DEC 78a], sous moniteur TOPS10 6.03 [DEC 78b], SAIL [HARVEY 74, FROST 75] et IRCAM [FROST 77]. C'est le plus rapide et le plus puissant système VLISP existant. Il est disponible dans les centres suivants : l'I.R.C.A.M. et le C.I.T.I.2 en France, et EDIMBOURG, TURIN et STANFORD SAIL à l'étranger.

2) le système VLISP-11 [GREUSSAY 79b], est réalisé sur l'ordinateur PDP 11 également de Digital Equipment Corporation [DEC 75] sous système RT11-V03 [DEC 78c]. Ce système fonctionne actuellement sur les processeurs PDP11 et sur les micro-processeurs LSI11 [DEC 78d] toujours sous système RT11-V03.

3) le système VLISP-8 [CHAILLOUX 79a], est conçu pour le micro-processeur Intel-8080 [INTEL 77a] et Zilog-80 [ZILOG 78]. Il est disponible sous le système de développement ISIS-II [INTEL 77b] et TRS80 Level II [TRS 78].

Chacun de ces systèmes représente une catégorie matérielle spécifique : gros système pour le PDP-10, mini-ordinateur pour le PDP-11 et micro-ordinateur pour le 8080. Les différences entre ces matériels sont considérables. Nous évoquerons les points suivants :

- la taille des mémoires. Le PDP10 permet d'accéder à 256k de 36 bits, le PDP11 à 28k de 16 bits *{Note 1}* et le 8080 à 64k de 8 bits.

- la vitesse d'exécution des instructions. Le transfert dans un registre du contenu d'un mot mémoire demande (indépendemment du nombre de bits transférés) 1.06 micro-sec dans un PDP10-KI, 3.2 micro-sec dans un PDP11/40 et 4.5 micro-sec dans un 8080-2Mhz.

- la puissance des instructions. Les jeux d'instructions de ces machines n'ont pas la même puissance ce qui amène à utiliser un nombre d'intructions plus ou moins grand. Par exemple pour la réalisation de l'allocation et de la construction d'un doublet de liste (le module CONS), il suffit de 2 instructions du PDP10 (qui occupent chacune un mot mémoire de 36 bits) alors qu'il faut un sous-programme de 9 instructions pour le PDP11 (chaque instruction nécessite de 1 à 3 mots de 16 bits) occupant un total de 12 mots pour l'appel et l'exécution du sous-programme. Quant aux performances du 8080 elles sont encore plus mauvaises puisqu'il faut un sous-programme de 21 instructions, qui nécessitent chacune de 1 à 3 mots de 8 bits) occupant un total de 30 mots mémoire.

{Note 1} ce nombre inhabituel de k mémoire (28k) provient du fait que les entrée/sorties sont réalisées en memory mapped I/O sur 4k mots, qui ajoutés aux 28k mémoire conduit bien à un espace adresse de 32k mots.

Malgrés leurs différences, il a été possible d'implanter sur toutes ces machines un système VLISP, tel celui que nous décrire dans cette étude. Ces différents systèmes ne se différencient que par des vitesses d'exécution et des espaces mémoire différents.

Enfin, le champ d'application du système VLISP est extrêmement vaste. Ses principales applications récentes recouvrent les domaines suivants :

- l'amélioration et la correction de programmes avec le système PHENARETE [WERTZ 79].
- la méta-interprétation de programmes récursifs avec le système CAN [GOOSSENS 79].
- les algorithmes d'unification [HULLOT 79].
- la synthèse de programmes à partir d'exemples avec le système SISP [JOUANNAUD 77].
- la compréhension de programmes avec le système RAINBOW [GREUSSAY 79a].
- la synthèse d'images colorées telle qu'elle est pratiquée par le groupe Art et Informatique de l'Université de Vincennes [AUDOIRE 76], [HUITRIC 76]
- l'aide à l'éducation des enfants retardés avec le système LOGO/LISP [WERTZ 79].
- les outils de conception de machines [CHAILLOUX 78b, CHAILLOUX 79b].

BIBLIOGRAPHIE

[ALLEN 78]
ALLEN J. : *The Anatomy of LISP,* Mc Graw Hill Inc., 1978.

[AUDOIRE 76]
AUDOIRE L. : *COLORIX : un périphérique de visualisation couleur,* Mémoire de maîtrise, UER Informatique, Université de Paris 8 - Vincennes, Juin 1976.

[BERKELEY 74]
BERKELEY E. C., BOBROW D. G. (editors) : *The Programming Langage LISP : Its Operation and Application,* The M.I.T. Press, Cambridge, Massachusets, 4th printing, March 1974.

[BOBROW 73]
BOBROW R. J., BURTON R. R., JACOBS J. M., LEWIS D. : *UCI LISP Manual,* Dept. of Information and Computer Science, University of California, Irvine, Technical Report #21 updated 10/73.

[BOLCE 68]
BOLCE J. F. : *LISP/360 : a Description of the University of WATERLOO LISP 1.5 for the IBM System 360,* 2nd ed., University of WATERLOO, Computer Centre, March 1968.

[CHAILLOUX 78a]
CHAILLOUX J. : *VLISP 8 un système LISP pour micro-processeur à mots de 8 bits,* RT 21-78, Département d'Informatique, Université de Paris 8 - Vincennes, Juillet 1978.

[CHAILLOUX 78b]
CHAILLOUX J. : *a VLISP interpreter on the VCMC1 machine,* LISP Bulletin #2, July 1978.

[CHAILLOUX 78c]
CHAILLOUX J. : VLISP *10 . 3 , Manuel de Référence,* RT 16-78, Université de Paris 8 - Vincennes, Août 1978.

[CHAILLOUX 79a]
CHAILLOUX J. : VLISP *8 . 2 , Manuel de Référence,* RT 11-79, Université de Paris 8 - Vincennes, Avril 1979.

[CHAILLOUX 79b]
CHAILLOUX J. : *Etude d'un compilateur* VLISP *optimisant,* Bulletin du Groupe Programmation et Langages, A.F.C.E.T., Division Théorique et Technique de l'Informatique, No. 9 pp. 26-36, Octobre 1979.

[CHAILLOUX 80]
CHAILLOUX J. : *Le modèle VLISP : description, implémentation et évaluation,* Thèse de 3ème cycle, La boratoire Informatique Théorique et Programmation, Université de Paris 6, Avril 1980.

[COILLAND 79]
COILLAND P., COLAITIS M. J. : *SIRLISP : Interprète LISP sur IRIS80,* E.N.S.T., Paris, Février 1979.

[DEC 75]
PDP 11 : Processor Handbook, Digital Equipment Corporation, Maynard Massachusetts, 1975.

[DEC 78a]
DECSYSTEM10 : *System Reference Manual,* Digital Equipment Corporation, Maynard Massachusetts, AD-09 16C-T1, March 1978.

[DEC 78b]
DECSYSTEM10 : *TOPS10,* Digital Equipment Corporation, Maynard Massachusetts, AD-09 16C-T1, March 1978.

[DEC 78c]
RT11 V03 : ***Documentation directory,*** Digital Equipment Corporation, Order No. DEC 11-ORDDB-A-D, Maynard Massachusetts, 1978.

[DEC 78d]
LSI 11 : ***Microprocessor Handbook,*** Digital Equipment Corporation, Maynard Massachusetts, 1978.

[DEC 78e]
VAX 11-780, Hardware/Software Handbook, Digital Equipment Corporation, Maynard Massachusetts, 1978.

[DURIEUX 78]
DURIEUX J. L. : ***TLISP, le système LISP de TOULOUSE,*** in Implémentation et Interprétation de LISP, Ecole IRIA, Toulouse, 1-3 Mars 1978.

[DURIEUX 79]
DURIEUX J. L., VIGNOLLE J. : ***Fausses Récursivités dans un Interprète LISP : une approche formelle de leur élimination,*** Université Paul Sabatier, Toulouse.

[FROST 75]
FROST M. : ***UUO Manual (Second Edition),*** SAILON 55.4, Stanford Artificial Intelligence Laboratory, July 1975.

[FROST 77]
FROST M. & HARVEY B. : ***The Stanford/IRCAM Monitor,*** Institut de Recherche et Coordination Acoustique/Musique, No. 2, October 1977.

[GARDNER 67]
GARDNER M. : ***Mathematical Games,*** Scientific American, pp. 124-125, March/April 1967.

[GOOSSENS 77]
GOOSSENS D. : ***CAN,*** Département d'Informatique, Université de Paris 8 - Vincennes, 1977.

[GOOSSENS 79]
GOOSSENS D. : *Meta-interpretation of Recursive List-processing Programs,* Proc. 6th I.J.C.A.I. pp. S7-S12, Tokyo, August 1979.

[GREENBLATT 74]
GREENBLATT R. : *The LISP Machine,* M.I.T. Artificial Intelligence Laboratory, Working Paper 79, November 1974.

[GREUSSAY 72]
GREUSSAY P. : *Manuel LISP 510 : description et utilisation,* Institut d'Intelligence Artificielle, Université de Paris 8 - Vincennes, NTP 2, Octobre 1972.

[GREUSSAY 75]
GREUSSAY P. : *LISP T1600 : Manuel de Référence,* Département d'Informatique, Université de Paris 8 - Vincennes, Février 1975.

[GREUSSAY 76a]
GREUSSAY P. : *VLISP : Structure et extension d'un système LISP pour mini-ordinateur,* RT 16-76, UER Informatique et Linguistique, Université de Paris 8 - Vincennes, Janvier 1976.

[GREUSSAY 76b]
GREUSSAY P. : *Descriptions compactes d'interprètes implémentables : une application au langage CONNIVER,* 2ème Colloque International sur la Programmation, ROBINET B. (éd.), Dunod, Paris, Avril 1976.

[GREUSSAY 76c]
GREUSSAY P. : *Iterative interpretations of tail-recursive LISP procedures,* Département d'Informatique, TR 20-76, Université de Paris 8 - Vincennes, Septembre 1976.

[GREUSSAY 77]
GREUSSAY P. : *Contribution à la définition interprétative et à l'implémentation des LAMBDA-langages,* Thèse, Université de Paris VII, Novembre 1977.

[GREUSSAY 78a]
GREUSSAY P. : *Communication personnelle,* **Octobre 1978.**

[GREUSSAY 78b]
GREUSSAY P. : *Le Système* VLISP *16,* Ecole Polytechnique, Décembre 1978.

[GREUSSAY 79a]
GREUSSAY P. : *Aides à la Programmation en LISP : outils d'observation et de compréhension,* Bulletin du Groupe Programmation et Langages, A.F.C.E.T., Division Théorique et Technique de l'Informatique, No. 9 pp. 13-25, Octobre 1979.

[GREUSSAY 79b]
GREUSSAY P. : VLISP*-11 Manuel de Référence, (à paraître),* Université de Paris 8 - Vincennes, 1979.

[HAFNER 74]
HAFNER C., WILCOX B. : *LISP/MTS programmer's guide,* Mental Health Research Institute, Michigan University, Ann ARBOR, January 1974.

[HAILPERN 79]
HAILPERN B. T., HITSON B. L. : *S-1 Architecture Manual,* Computer System Laboratory, Stanford Electronics Laboratories, Stanford University, Technical Report No. 161, STAN-CS-79-715, January 1979.

[HARVEY 74]
HARVEY B. : *Monitor Command Manual (Second Edition),* SAILON 54.4, Stanford Artificial Intelligence Laboratory, September 1974.

[HEARN 69]
HEARN A. C. : *Standard LISP,* in Bobrow D. G. (ed.) : LISP Bulletin, ACM SIGPLAN Notices, Vol. 4 no. 9, September 1969.

[HEARN 73]
HEARN A. C. : *REDUCE 2 User's Manual,* Utah Computational Physics group UCP-19, Second Ed., March 1973.

[HEARN 74]
HEARN A. C. : *REDUCE 2 Symbolic Mode Primer,* Utah Computational Physics group Operating Note 5.1, October 1974.

[HOLLOWAY 80]
HOLLOWAY J., STEELE G., SUSSMAN G., BELL A. : *The SCHEME 79 Chip,* M.I.T. Artificial Intelligence Laboratory, AI Memo No. 255, Draft, January 1980.

[HORNING 79]
HORNING J. J. : *Additionnal Viewpoints on the History of Programming Languages Conference,* Annals of the History of Computing, Vol. 1 pp. 69-71, July 1979.

[HUITRIC 76]
HUTRIC H. : *Couleur et calcul : calcul de la couleur,* Thèse de 3ème cycle, Université de Paris 8, 1976.

[HULLOT 79]
HULLOT J. M. : *Associative Commutative Pattern-matching,* Proc. 6th I.J.C.A.I. pp. 406-412, Tokyo, August 1979.

[INTEL 77a]
8080 : *Assembly Language Programming Manual,* ON : 98-00301B, Intel Corporation, 1977.

[INTEL 77b]
MDS : *ISIS II System user's guide,* ON : 98-306B, Intel Corporation, 1977.

[JOUANNAUD 77]
JOUANNAUD J. P. : *Sur l'inférence et la synthèse automatiques de fonctions LISP à partir d'exemples,* Thèse, Université Pierre et Marie Curie (Paris 6), Novembre 77.

[KNIGHT 74]
KNIGHT T. : *The CONS Machine,* M.I.T. Artificial Intelligence Laboratory, Working Paper 80, November 1974.

[KNUTH 69]
KNUTH D. E. : *The Art of Computer Programming, Vol. 1 : Fundamental Algorithms,* Addison-Wesley, Reading, Massachusetts, 2nd printing, 1969.

[LAUBSCH 76]
LAUBSCH J. H., KRAUSE D., HESS K., SCHATZ W. : *MACLISP Manual,* CUU-Memo-3, Universitat Stuttgart, Stuttgart, Mai 1976

[LECOUFFE 77]
LECOUFFE P. : *étude et définition d'une machine langage LISP,* Thèse de spécialité, Université des sciences et techniques de LILLE, Décembre 1977.

[LISPMACHINE 77]
LISP MACHINE GROUP, BAWDEN A., GREENBLATT R., HOLLOWAY J. KNIGHT T., MOON D., WEINREB D. : *LISP Machine Progress Report,* M.I.T. Artificial Intelligence Laborartory, Memo No. 444, August 1977.

[LUX 78]
LUX A. : *LISP - IRIS 80 : Manuel d'Utilisation,* Laboratoire IMAG, Février 1978.

[MACSYMA 75]
MACSYMA Reference Manual, Project MAC Mathlab Group, M.I.T., November 1975.

[MARTI 79]
MARTI J., HEARN A. C., GRISS M. L., GRISS C. : *STANDARD LISP REPORT,* ACM SIGPLAN Notices, Vol. 14 No. 10, October 1979.

[McCARTHY 60a]
McCARTHY J. : *LISP 1 Programmer's manual,* M.I.T. Computation Center & Research Laboratory of Electronics, Cambridge Mass., March 1, 1960.

[McCARTHY 60b]
McCARTHY J. : *Recursive Functions of Symbolic Expressions and their Computation by Machine Part I,* C.A.C.M., vol. 3, March 1960.

[McCARTHY 62]
McCARTHY J., ABRAHAMS P. W., EDWARDS D. J., HART T. P., LEVIN M. I. : *LISP 1.5 Programmer's manual,* the M.I.T. Press, Cambridge, Mass., 1962.

[McCARTHY 78]
McCARTHY J. : *History of LISP,* History of Programming Languages Conference, ACM SIGPLAN Notices, Vol. 13 No. 8, 1978.

[MEAD 80]
MEAD C., CONWAY L. : *Introduction to VLSI systems,* Addison-Wesley publishing company, 1980.

[MOON 74]
MOON D. A. : *MACLISP Reference Manual,* M.I.T. Project MAC, Cambridge Mass., April 1974.

[MOORE 76]
MOORE J. S. : *the INTERLISP Virtual Machine Specification,* XEROX Palo Alto Research Center, Palo Alto Cal., September 1976.

[NIVAT 79]
NIVAT M. : *La Recherche en Programmation et ses Moyens de Calcul,* La Gazette du L.I.T.P. Bulletin No. 9, Janvier 1979.

[PERROT 79]
PERROT J. F. : *LISP et lambda-calcul,* Lambda-Calcul et Semantique formelle des langages de programmation B. Robinet, Ed., pp. 277-301, LITP-ENSTA, Paris, 1979.

[QUAM 72]
QUAM L. H., DIFFIE W. : *Stanford LISP 1.6 Manual,* SAILON 28.6, Computer Science Dpt, Stanford University, 1972.

[ROBINET 78]
ROBINET B. : *petit précis de λ calcul,* in Implémentation et Interprétation de LISP, Ecole IRIA, Toulouse, pp. 15-24, 1-3 Mars 1978.

[SCHOETTL 75]
SCHOETTL J. E., WERTZ H. : *Des dragons à foison,* Artinfo/Musinfo #21, Groupe Art et Informatique, Université de Paris 8 - Vincennes, 1975.

[SHIMADA 76]
SHIMADA T. ,YAMAGUSHI Y., SAKAMURA K. : *A LISP Machine and Its Evaluation,* Systems Computers Controls, Vol. 7, No. 3, 1976 (translated from Denshi Tsushin Gakkai Rombunshi, Vol. 59-d, No. 6 June 1976, pp. 406-413).

[STEELE 79]
STEELE G. L. Jr., SUSSMAN G. J. : *Design of LISP-Based Processors or, SCHEME: A Dielectric LISP or, Finite Memories Considered Harmful or, LAMBDA: The Ultimate Opcode,* AI Memo No. 514, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., March 1979.

[STOYAN 78a]
STOYAN H. : *LISP,* Thèse, Technishe Universitat Dresden, Dresden, RDA.

[STOYAN 78b]
STOYAN H. : *LISP-Programmier Handbuch,* Akademie Verlag, Berlin, 1978.

[TAFT 79]
TAFT S. T. : *The Design of an M6800 LISP Interpreter,* BYTE, Vol. 4 No. 8 pp. 132-152, August 1979.

[TAKI 79]
TAKI K., KANEDA Y., MAEKEWA S. : *The Experimental LISP Machine,* Proc. 6th I.J.C.A.I., TOKYO, 1979.

[TEITELMAN 73]
TEITELMAN W. : *CLISP - Conversationnal LISP,* Proc. 3rd I.J.C.A.I., pp. 686-690, Stanford, California, August 1973.

[TEITELMAN 75]
TEITELMAN W. : *INTERLISP Reference Manual,* 2nd revision, Xerox Palo Alto Research Center, December 1975, 3rd revision, October 1978.

[TEITELMAN 77]
TEITELMAN W. : *A Display Oriented Programmer's Assistant,* Proc. 5th I.J.C.A.I., pp. 905-915,Cambridge, Mass., August 1977.

[TRS 78]
TRS80 : *LEVEL II,* Radio Shack, Ft. Worth, Texas 76102

[WEINREB 79]
WEINREB D. & MOON D. A. : *LISP MACHINE MANUAL,* M.I.T. Cambridge, Mass., January 1979.

[WEISSMAN 67]
WEISSMAN C. : *LISP 1.5 Primer,* Dickenson Publishing Company Inc., Belmont, California, 1967.

[WERTZ 74]
WERTZ H. : LISP CAB500, *Rapport de Recherche goupe II équipe 9,* Université de Paris 8 - Vincennes, 1974-1975.

[WERTZ 77]
WERTZ H. : *VLISP - AID,* Université de Paris 8 - Vincennes, RT 10-77, Juillet 1977.

[WERTZ 78]
WERTZ H. : ***Un système de compréhension, d'amélioration et de correction de programmes incorrects,*** Thèse de 3ème cycle, Université Paris VI, Juillet 1978.

[WERTZ 79]
WERTZ H. : ***Computer Aided Education for Mentaly Retarded Children,*** Proc. International Symposium on Computer and Education, Dusseldorf, RFA, Mars 1979.

[WHITE 76]
WHILE J. L. : ***Mail from JONL at MIT-ML to LISP-discussion at MIT-AI,*** February 1976.

[WHITE 78]
WHITE J. L. : ***Programm is Data,*** History of Programming Languages Conference, ACM SIGPLAN Notices, Vol. 13 No. 8 pp. 217-223, 1978.

[WINSTON 77]
WINSTON P. H. : ***Artificial Intelligence,*** Addison-Wesley Company Inc., 1977.

[ZILOG 79]
Z80 : CPU Technical Manual, Zilog Inc., 1979.

INDEX BIBLIOGRAPHIQUE

NOTES SUR L'OBSERVATION ET LA COMPREHENSION DE PROGRAMMES

Patrick GREUSSAY

Département d'Informatique
Université Paris VIII - Vincennes
75571 PARIS Cédex 12
&
C.N.R.S. LA 248 L.I.T.P.
2 place Jussieu
75005 Paris

Resumé :

Un important courant de recherche en Intelligence Artificielle met aujourd'hui l'accent sur le développement d'outils visant à automatiser partiellement la ***compréhension*** de programmes. Ce courant se situe actuellement à la croisée de trois directions de recherches très actuelles : théorie de la programmation, développement d'outils de mise au point interactive, interrogation sur les processus psychologiques mis en jeu dans l'activité des programmeurs experts. Ces recherches doivent déboucher sur l'automatisation partielle de la documentation de systèmes ainsi que sur des outils très puissants d'aide au développement de grands programmes : correction et maintenance. Il paraît indispensable, en préalable à la mise en place de systèmes de compréhension automatisés, de recueillir et de systématiser des données concernant le comportement de programmeurs experts, en situation de compréhension et d'analyse de programmes. Nous décrirons une expérience de compréhension dont l'analyse des résultats indique la difficulté, pour les programmeurs, de manipuler des représentation multiples sans outils appropriés d'aide à la programmation. Nous conclurons par l'examen d'un système de compréhension automatique, utilisant un ensemble de représentations multiples pour un même programme, et fondé sur un ensemble de régles de traductions entre représentations récursives et représentations linéaires.

MOTS-CLES:

mise au point, VLISP, programmation interactive, système RAINBOW, compréhension automatique, représentations multiples, psychologie de la programmation.

1.0 Introduction

La recherche en Intelligence Artificielle est essentiellement définie par la programmation effective de certains modèles de comportement, pour une grande part d'ordres linguistiques, conceptuels et sensori-moteurs.

La mise en oeuvre effective de ces programmes permet de tester et de falsifier très rapidement ces modéles : la construction et la vérification d'hypothèses s'apparentant très fortement, et allant même jusqu'à s'identifier avec le processus de mise au point de programmes. Des structures de données et de contrôle inédites doivent ainsi pouvoir être ***très rapidement*** mises en oeuvre, structures dont l'ensemble constitue ce qu'on pourrait nommer des ***langages provisoires***, très dépendantes du modèle de comportement que la construction du programme d'Intelligence Artificielle permet de valider. Le modèle se révélant en premiere approximation inadéquat, partiellement ou totalement, la correction ou la reconsidération du modèle induisent alors l'abandon ou la modification drastique des structures évoquées plus haut. Elles conduisent dans des délais très brefs à la conception et l'implémentation de nouveaux objets pour lesquels le même processus se rèitére jusqu'à stabilisation du modèle.

C'est très naturellement dans ce type de ***programmation instable*** dont la rapidité est encore accentuée par sa mise en oeuvre interactive que se révèlent les plus nécessaires des outils-systèmes de développement, visant à rendre automatiques la ***compréhension***, ***documentation*** et ***maintenance*** de programmes. De tels systèmes doivent rendre automatiques, partiellement ou totalement, la détection et la correction des innombrables micro-erreurs inhérentes à la très grande taille des programmes développés en mode interactif.

Les systèmes de compréhension de programmes doivent permettre de faire coexister plusieurs représentations du même programme en développement : ces systèmes doivent jouer le rôle d'outils de documentation incrémentale, d'outils de ***lecture*** d'intentions de programmes et de production automatique de leurs propriétés.

Nous faisons l'hypothése qu'il doit exister une continuité entre le données qui seront livrées par l'observation systématique du comportement de programmeurs experts en situation de compréhension et les nouveaux outils, dits de compréhension automatique de programmes qui commencent à apparaître: dans les deux cas, il s'agit de rendre ***visibles*** des comportements en rendant explicites les représentations sur lesquelles ils sont fondés. Un système de compréhension automatique original sera à cette occasion présenté, le système RAINBOW (Greussay 1979). Nous examinerons également, à partir d'un cas concret, les difficultés liées à la compréhension automatique lorsqu'on la compare à celle mise en jeu par des programmeurs.

Le langage de programmation LISP est actuellement le support massif de telles expérimentations, c'est pourquoi la plupart de nos exemples seront exprimés en langage VLISP 10 (CHAILLOUX 1979) dont l'environnement de programmation très complet illustre bien la nature des outils d'aide à la programmation qu'on peut attendre d'un système LISP contemporain.

2.0 De l'observation à la compréhension

L'expérience de la programmation en Intelligence Artificielle doit nous amener à reconsidérer le point de vue actuel qui consiste à rejeter la mise au point de programmes et ses outils, au profit d'une validation préalable par preuve de la correction, partielle ou totale, de ses spécifications à différents niveaux d'abstractions.

Outre que la faillibilité n'en est certes pas absente (GERHART 1976), que les erreurs de programmation constituent en elles-mêmes une source de données considérable et encore fort mal connue sur les processus intellectuels mis en jeu en programmation (SUSSMAN 1974), la pratique de la programmation en Intelligence Artificielle ne nous permet pas de mésestimer l'importance de la mise au point (De MILLO 1979).

En programmation à long terme, le plus souvent interactive, parfois quasi-improvisée au terminal, en tout cas profondément incrémentale, la validation préalable des programmes ne semble pas être une activité prioritaire. Nous avons plus urgemment besoin d'un ensemble cohérent d'outils de mise au point de programmes, ensemble totalement intégré

1) aux interprètes qui interprètent ces programmes sous divers modes

2) aux éditeurs (CICCARELLI 1978) qui en permettent l'introduction et la modification.

Dans les cas simples, il semble qu'en programmation interactive, les étapes de mise au point, d'essais, de modifications, voire de commentaires de fragments de programmes peuvent être anticipées, et en quelque sorte comprises et résumées par de tels systémes. Les outils classiques de mise au point en LISP permettent d'observer le comportement des programmes. Pouvons-nous les étendre à l'observations des comportements des programmeurs? Un tel système d'aide à la mise au point pourrait fournir, de surcroît à sa fonction d'aide à la programmation, cet instrument d'observation et d'enregistrement de l'activité d'un programmeur au cours de sessions interactives. Nous faisons l'hypothèse qu'il existe des régularités observables dans les cycles de mise au point.

Reste que les outils classiques d'observations en LISP (traces et intervenants de type ADVISE), s'ils ont une puissance opérationnelle considérable, laissent leur utilisateur devant un vide sémantique total. Pour passer de l'observation à la compréhension, de nombreuses questions doivent être considérées:

1) concernant les erreurs de programmation: comment étudier et classifier les erreurs? quelles erreurs doit-on et peut-on capter? quelles erreurs sont anticipables (WERTZ 1979)?

2) concernant la compréhension de programmes: comment rendre explicite le fonctionnement d'un programme correct ou non? comment automatiser la compréhension de programmes simples: un programme LISP d'une certaine ampleur sera composé en majeure partie d'un trés grand nombre de fonctions auxiliaires très courtes; l'expérience indique que la plupart des erreurs difficiles à déceler interviennent dans le détail de rédaction de ces fonctions auxiliaires. Comment rendre ces erreurs évidentes en livrant automatiquement l'intention de ces fonctions au fur et à mesure de leur frappe au terminal?

3.0 Représentations, Compréhension, Mise au point

Que signifie l'énoncé *concevoir un programme* ?

Voici une réponse partielle issue de la pratique de la programmation en Intelligence Artificielle : concevoir un programme, c'est en mettre au point un autre déjà connu (SUSSMAN 1975) . Ce point de vue a été repris récemment dans une perspective plus théorique par (DERSHOWITZ 1976).

Le point de vue de la mise au point implique qu'en un certain sens, un programme n'est ni juste ni faux. Ni juste, car toujours susceptible d'être modifié, amélioré ou étendu, à mesure que surgissent de nouveaux besoins ou que s'approfondit la compréhension du problème dont le programme est le modèle. Ni faux, car toujours susceptible d'être tracé et édité.

Ainsi le point de vue de la mise au point met l'accent sur l'ubiquité des modifications de programmes, et donc des modifications de représentation et de compréhension.

Une expérience de compréhension

En 1977-1978, nous avons tenté de repérer concrètement quelques unes des difficultés de la compréhension de programmes, à travers une expérience menée sur un cas très simple.

On notera que des expériences de ce type ne visent à rien d'autre qu'à ***apprendre à recueillir des données***, des observables des processus psychologiques mis en jeu en programmation. Ces processus sont encore trop mal connus pour fixer prématurément une systématisation de telles observations.

Nous avons soumis à plusieurs programmeurs de haut niveau la fonction suivante, avec pour mission d'en découvrir l'intention.

```
                   (DE SKE (L R) (COND
                      ((ATOM L) NIL)
2 ----------------    ((MEMQ L R) T)
3 ----------------    ((SKE (CAR L) (CONS L R)) T)
4 ----------------    (T (SKE (CDR L) (CONS L R)))))
```

L'expérience consistait à présenter la fonction sans autre commentaire en laissant les sujets réfléchir quelques minutes, puis à compléter cette fonction par des ***thèmes*** présentés successivement, certains très intuitifs, d'autres plus formels, éléments de connaissance susceptibles d'en faciliter la compréhension.

Thème 1 : L'appel initial de SKE est

(SKE *une-liste*-L NIL)

Thème 2 : Les deux arguments de la fonction standard MEMQ peuvent être des listes.

Thème 3 : SKE décèle des ***effets de bord***, antérieurs à son application

Thème 4 : L est une liste circulaire

Thème 5 : Soit x et y deux cellules de liste, et la relation

x << y

si y = (x . z) ou (z . x)
ou
si x << (CAR y) ou x << (CDR y)

Une liste est non-circulaire si, pour deux de ses cellules x et y on n'a jamais simultanément

x << y et y << x

Un seul des sujets participant à l'expérience découvrit la destination de SKE dès la présentation du thème 3. La présentation du thème 5 fut nécessaire à l'un d'entre eux. Tous les autres sujets donnèrent la réponse correcte lors de la présentation du thème 4.

Quels furent les éléments observés?

Le thème 3 indique déjà nettement que L ou une de ses sous-listes est circulaire: une liste circulaire doit être construite par la mise en jeu des primitives RPLACA et RPLACD. Le premier sujet ayant décelé l'intention de SKE programmait depuis quelques jours une routine d'impression élégante de listes circulaires.

Le thème 4 est positif, le thème 5 est négatif: il indique ce qu'une liste circulaire *n'est pas*. Le sujet ayant attendu le thème 5 pour livrer une réponse correcte admit volontiers qu'après l'avoir considéré un instant, il perdit de vue le fait que la fonction SKE est booléenne.

Tous les sujets firent cependant plusieurs observations pertinentes au cours de l'expérience. La clause 4 fut déclarée par tous itérative, opérant le balayage de L dans le sens des CDRs, et la clause 3 récursive, balayant L dans le sens des CARs. L'analogie avec le schéma de la fonction EQUAL fut unanimement constatée.

Le thème 1 semble avoir provoqué un brouillage de représentations: la variable R fut bien reconnue comme un accumulateur, la représentation statique de son contenu a fait, de l'avis général, probléme. Une fois l'accent mis sur l'aspect dynamique de sa construction, la destination de R est devenue pour chacun évidente.

Qu'en conclure ?

L'expérience a mis en valeur des difficultés conceptuelles liées à des *interférences* de représentations

1) interférence entre le schéma récursif de SKE et la représentation itérative qu'entraîne la mise en jeu d'un accumulateur: la variable R retient tous les CDRs de toutes les sous-listes de L, mais R est testée dans la condition d'arrêt.

2) interférence entre construction et rétention. CONS construit une ***nouvelle*** structure, MEMQ teste à la clause 2 l'identité des deux listes: L la liste courant examinée, et une sous-liste de L recueillie dans R à une étape ***précédente*** du balayage.

Cette interférence a fait perdre de vue à nos sujets que MEMQ utilise EQ, qui teste l'identité ***physique*** de deux objets, indifférement atomes ou listes, par comparaison d'adresses. Ce test d'identité physique a rendu vaine la tentative d'analyse, par nos sujets, de la structure interne des sous-listes de R.

De plus une interférence de représentations spatiales a dû jouer: entre le caractère arborescent de L, impliqué par le schéma de type EQUAL de SKE, et le caractère linéaire de R, construite par CONS successifs.

3) enfin interférence de type manipulatoire: L est ***décomposée*** en (CAR L) et (CDR L) au cours du balayage. Mais L ***n'est pas*** décomposée lors de sa composition par CONS dans R.

4.0 Un système de compréhension automatique

L'étude précédente nous a indiqué quelques-unes des difficultés liées à la compréhension de programmes dont l'intention est inconnue. Nous examinerons à présent quelques unes des possibilités offertes par un système de compréhension automatique de programmes spécialisé dans la lecture d'une classe restreinte mais très utilisée de fonctions de construction en LISP. Le système RAINBOW (GREUSSAY 1979), totalement interactif, sert de système frontal à VLISP 10. Il permet la traduction immédiate des fonctions introduites au terminal dans deux types de notations formelles, exprimant l'une le calcul récursif effectué par ces fonctions, l'autre le résultat générique livré à la suite d'appels symboliques. Nous ne décrirons pas ici le processus de transformation des appels symboliques en ***visualisation d'intention*** mais nous l'illustrerons par des exemples d'utilisation pratique.

Comment rendre ***visible*** l'intention d'une fonction de construction auxiliaire?

Nous introduirons à cet effet deux notations indicielles

1) linéaire

Elle sera utilisée pour exprimer les résultats génériques ou ***intentions*** des fonctions analysées: elle exprime des caractéristiques de séquences.

Cette notation est une spécialisation de celles employée dans (GOOSSENS 1979).

$$[\overset{n}{\underset{1}{i}}\ E_i] \ =df\ (E1\ E2\ \ldots\ En)$$

EXEMPLES:

$$(CAR\ [\overset{n}{\underset{1}{i}}\ E_i])\ ☞\ E1$$

$$(CDR\ [\overset{n}{\underset{1}{i}}\ E_i])\ ☞\ [\overset{n}{\underset{2}{i}}\ E_i]$$

$$(CONS\ E1\ [\overset{n}{\underset{2}{i}}\ E_i])\ ☞\ [\overset{n}{\underset{1}{i}}\ E_i]$$

$$(DCONS\ X\ [\overset{n}{\underset{1}{i}}\ E_i])\ ☞\ [\overset{n}{\underset{1}{i}}\ [X\ .\ E_i]]$$

2) récursive

Elle sera utilisée pour exprimer le calcul récursif effectué par la fonction. Elle prendra la forme générale:

```
n_______
|       |
i ( ... - ... ) expression-d'arrêt
|       |_______|
1
```

qui indique *n* niveaux d'imbrications d'appels s'achevant sur l'expression d'arrêt (*n* étant la longueur de la liste qui sera la valeur de la variable de récursion).

EXEMPLE

Le système RAINBOW traduit, en mode interactif, la définition de fonction suivante

```
(DE append (X Y)
    (IF (NULL X) Y
        (CONS (CAR X)
              (append (CDR X) Y))))
```

sous la forme schématique

```
n___________
|           |
i (CONS Xi  -) Y
|           |__|
1
```

qui résume l'expression imbriquée

```
(CONS X1 (CONS X2 ... (CONS Xn Y) ... ))
```

```
                            i-1
avec Xi  =df  (CAR (CDR        X))

  n
  |
 [j Xj]  =df  X
  |
  i

  n
  |
 [j Xj]  =df  (CDR X)
  |
  i+1
```

Voici deux exemples d'utilisations interactives de RAINBOW en documentation automatique de programmes, le premier assez simple, le second nettement plus complexe.

EXEMPLE 1:

L'utilisateur introduit la définition suivante:

```
(DE f (X Y)
    (IF (NULL X) NIL
        (APPEND (g (CAR X) Y)
                (f (CDR X) Y))))
```

que le systéme traduit immédiatement en

```
n___________________
|                  |
i (APPEND (g Xi Y) -) NIL
|                  |__|
1
```

puis la définition suivante

```
(DE g (A Y)
    (IF (NULL Y) NIL
        (CONS (LIST A (CAR Y))
              (g A (CDR Y)))))
```

traduite alors en

```
n______________
|             |
i (CONS [A Yi] -) NIL
|             |__|
1
```

puis l'utilisateur livre à RAINBOW l'appel symbolique

```
(f (*list* A) (*list* B))
```

qui se résout sous la forme linéaire

```
 n m
 | |
[i j [Ai Bj]]
 | |
 1 1
```

qui révèle l'intention de la fonction *f*:
f construit le ***produit cartésien*** de ses deux listes-arguments.

Le processus de résolution sous forme linéaire utilise un ensemble de régles de réduction de représentations récursives en des représentations linéaires.

Celles qui ont été mises en jeu dans l'exemple précédent sont:

- la définition de la fonction *g* elle-même.

- la régle RC1:

```
n__________                  n
|          |                 |
i (CONS Ei - ) NIL    ☞     [i Ei]
|          |__|              |
1                            1
```

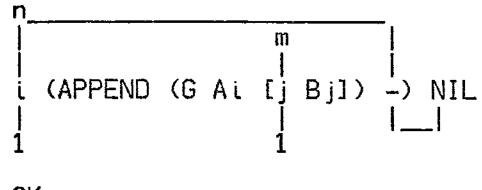
- la régle RA1:

```
n______________                  n
|              |                 |
i (APPEND [Ei] - ) NIL    ☞     [i Ei]
|              |__|              |
1                                1
```

Le système RAINBOW livre ainsi successivement :

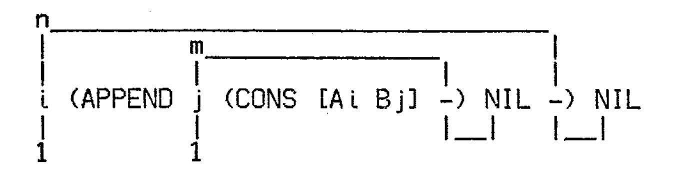
? (f (*list* A) (*list* B))

Les lignes précédées de "?" sont introduites par l'utilisateur



OK

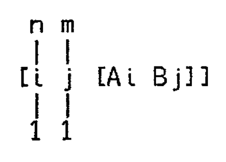
? ap all
APPLYING... G GIVING...



APPLYING... RC1 GIVING...

```
n_______________________
|             m         |
|             |         |
i (APPEND [j [Ai Bj]] -) NIL
|             |         |__|
1             1
```

APPLYING... RA1 GIVING...



La forme linéaire obtenue

OK

EXEMPLE 2:

Cherchant à visualiser l'effet d'une récursion assez complexe, l'utilisateur introduit au terminal:

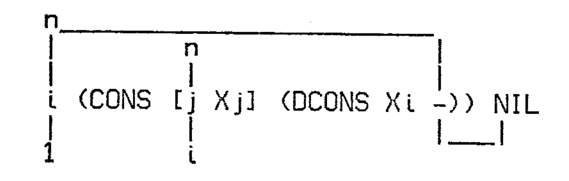
```
?  (DE vecmat (E)
       (IF (NULL E) NIL
           (CONS E (DCONS (CAR E)
                          (vecmat (CDR E))))))

   n_________________________
   |         n               |
   |         |               |
   i (CONS  [j  Ej] (DCONS Ei -)) NIL
   |         |               |___|
   1         i

  OK

?  cl (vecmat (*list* X))
```

puis un appel symbolique



```
   n_________________________
   |         n               |
   |         |               |
   i (CONS  [j  Xj] (DCONS Xi -)) NIL
   |         |               |___|
   1         i
```

```
  OK
?  ap all
  APPLYING... RDC4 GIVING...

   n______________________________
   |       i-1_________      n     |
   |        |          |     |     |
   i (CONS  j  (CONS Xj -)  [k Xk] -) NIL
   |        |          |__||       |__|
   1        1              i

  APPLYING... RC1 GIVING...

   n i-1_________      n
   | |           |     |
  [i j  (CONS Xj -)   [k Xk]]
   | |           |__||
   1 1               i

  APPLYING... RC2 GIVING...

   n   i-1  n
   |    |   |
  [i  [j Xj k  Xk]]
   |    |   |
   1    1   i

  APPLYING... RIND1 GIVING...

   n  n
   |  |
  [i [j Xj]]
   |  |
   1  1

  OK
```

La forme linéaire finale

5.0 Conclusion

L'examen des nouveaux outils d'observation et de compréhension disponibles en LISP nous amène à infléchir notre pensée sur la programmation: comme le suggére (WINOGRAD 1979) il semble aujourd'hui assez net que l'activité principale en programmation n'est plus essentiellement la création de programmes nouveaux, mais bien plutôt l'*intégration*, la *modification* et l'*élucidation* de programmes déjà existants. Par ailleurs de tels outils donnent à penser que la notion de langage de programmation n'est plus aussi nette qu'auparavant, en ceci que la distinction entre langage et environnement de programmation n'est plus clairement fixée. Nous attendons des développements à venir qu'ils permettent la mise en jeu, pour le même programme, de représentations multiples, autorisant la programmation directe par manipulation de représentations très proches de l'intuition du programmeur. Cette intuition peut varier au cours du développement de programmes d'une certaine ampleur. Nous attendons des nouveaux systèmes qu'ils suivent au plus près cette intuition variable, en rendant interactivement cette intuition *visible*.

Dans cette perspective, le problème de la mise au point nous paraît être de nature essentiellement *interprétative*: concernant moins le texte statique d'un programme qu'un ensemble très mal connu de processus intellectuels. Nous aurons à l'avenir besoin d'un instrument qui nous permette d'*observer* ces processus. Les données recueillies par un tel instrument peuvent nous permettre d'inférer la structure interprétative qui manipule, modifie et combine les représentations de connaissances en programmation qui sont mises en jeu implicitement par les programmeurs.

6.0 Bibliographie

[1] CHAILLOUX J. *VLISP 10.3, Manuel de Référence* Université Paris-8-Vincennes, Août 1978

[2] CICCARELLI E. *An Introduction to the EMACS Editor* M.I.T. Artificial Intelligence Memo 447, January 1978

[3] De MILLO R.A., LIPTON R.J., PERLIS A.J. *Social Processes and Proofs of Theorems and Programs* Comm. ACM, Vol 22, no 5, May 1979, 271-280

[4] DERSHOWITZ N., MANNA Z. *The Evolution of Programs : A System for Automatic Program Modification* Stanford Artificial Intelligence Laboratory, Memo AIM-294, December 1976

[5] GERHART S.L., YELOWITZ L. *Observations of Fallibility in Applications of Modern Programming Methodologies*, IEEE Transactions on Software Engineering, Vol.SE-2, no 3, September 1976, 195-207

[6] GOOSSENS D. *Meta-interpretation of Recursive List-processing Programs* I.J.C.A.I 1979, August 20-23, Tokyo, s7-s11

[7] GREUSSAY P. *How to Use the RAINBOW System* Université Paris-8-Vincennes, Département d'Informatique, RT-79-5, Mars 1979

[8] SUSSMAN G.J. *The Virtuous Nature of Bugs* Proc. of AISB Summer Conference, Brighton, July 1974, 224-237

[9] SUSSMAN G.J. *A computer model of skill acquisition* American Elsevier, N.Y., 1975

[10] WERTZ H. *Automatic Program Debugging* I.J.C.A.I 1979, August 20-23, Tokyo, 951-953

[11] WINOGRAD T., *Beyond Programming Languages*, C.A.C.M., Vol. 22, no. 7, July 1979, 391-401

PROGRAM UNDERSTANDING BY REDUCTION SETS

Patrick GREUSSAY

Departement d'Informatique
Universite Paris VIII - Vincennes
75571 PARIS Cedex 12
&
C.N.R.S. LA 248 L.I.T.P.
2 place Jussieu
75005 Paris

Abstract :

While building or understanding large LISP systems, many small auxiliary functions are often subject to errors or misunderstanding, in the case of very involved recursions. RAINBOW is a specialized program understanding system able to reduce automatically such sets of recursive functions to a form where the goal of these sets are clearly displayed. RAINBOW can display interactively the goal-forms into two sets of new external 2-dimensional notations: recursive and linear. Program understanding is obtained by the translation of the original set of LISP functions into the open recursive notation, then by elementary symbolic evaluation yielding closed linear forms of the original functions. Those linear forms are exactly the goals wanted. RAINBOW operates efficiently on a definite class of LISP functions, and uses an extendable set of reduction rules, which constitute the symbolic interpreter. RAINBOW can be used interactively if a user want to verify that a set of functions perform its intended goal, or can be incorporated easily as a specialized component of a larger program understanding system. This paper shows how RAINBOW operates on sets of recursive functions building combinatorial objects.

KEY-WORDS:

automatic program understanding, program debugging, VLISP, RAINBOW system, multiple representations, program transformation, symbolic interpretation.

1. INTRODUCTION

RAINBOW is a specialized interactive program understanding system. It asks from its user a set of recursive functions definitions, then extracts and displays graphically its *goal*, in terms of properties of lists viewed as sequences.

To display the goal of a definition set, we have introduced two classes of 2-dimensional external notations which are implemented within RAINBOW. The *open* notation is a compact notation for recursive programs or data structures, the *closed* notation expresses intrinsic properties of linear sequences in term of generic properties of their elements.

Program understanding is obtained by the translation of the original set of functions into the open notation, then by elementary symbolic evaluation yielding closed forms of the original functions. The closed forms are exactly the goals wanted.

Presently, RAINBOW can reason about classes of data-structures as lists considered as sequences, extensions to recursive LISP functions operating on other classes of data-structures are considered.

RAINBOW can be used either as a front-end of a LISP system, or as a specialized part for low-level understanding of sequences, in a larger program understanding system. An analogy with low-level machine vision is here in order: scene analysis has to rely upon intrinsic properties of pictures, as incidence, gradient, illumination or texture. We believe that a large program understanding system must also rely upon intrinsic properties of the data involved in the programs tentatively analyzed.

A programming apprentice system (RICH 1979, WERTZ 1979), if used interactively, must have the capability of focusing in a visually understandable way, to lower levels of plans. The plans diagrams described in (SHROBE 1979) seem promising either at very high level of inter-module analysis, or when the goal of a module is intermixed with other goals, or when global side-effects are involved. However these plans, displaying very well the global course of actions involved in analysed programs, do not seem appropriate to be used for lower-level modules because they do not allow to display the structure of the elementary goals for simple modules. Unfortunately, in very large systems, it seems that the simplest modules are the most error-prone.

RAINBOW allows the user to check immediately, in a visually understandable way, the goal of a set of functions definitions. The user does not have to provide any assertion to verify ,about his definitions set, because in a sense RAINBOW is precisely reducing this set to its assertion.

RAINBOW is an implemented system written in VLISP (CHAILLOUX 1978) running on DEC KI-10, and PDP 11/40. The external notations can be visualized on any kind of display or hard-copy terminal.

2. OVERVIEW OF THE RAINBOW SYSTEM

The symbolic interpreter:

The symbolic interpreter is essentially a production system having an initial fixed set of reduction rules for the handling of LISP sequences. When a user submits a new function definition to RAINBOW, the reduced closed linear form obtained as a goal is incorporated by the system into the set of rules. The interpreter is able to expand every inner function call with the replacement part of the corresponding rule along with the renaming of variables when necessary (α-conversion). As in (BOYER 1977), The interpreter operates iteratively until no more rule can apply to the reduced form.

The definitions entered can be also called within the LISP iterpreter, and every step of the reduction can be interactively reversed, providing an history of the reduction. Also at user-level, RAINBOW can handle symbolic function calls.

The process of resolution into linear forms uses a set of reduction rules for the translation of recursive representations into linear representations.

The reduction set of rules for a LISP function is expressed into two new classes of graphical notations for recursive programs and data, and for linear sequences.

Recursive and linear external notations:

1) Linear

It is used to express generic results or *goals* of the analysed functions: it expresses characteristics of sequences.

This notation is a 2-dimensional specialization of the one used in (GOOSSENS 1979).

$$[\overset{n}{\underset{1}{i}}\ E_i] \ \text{=df}\ (E_1\ E_2\ \ldots\ E_n)$$

The letter *i* is an "index variable" ranging from lower to upper indices, here from *1* to *n*. When one of the indices is 0, the notation denotes the empty sequence. In the context of RAINBOW, the range of the index variable is the LISP expression immediately following it.

EXAMPLES:

$$(\text{CAR}\ [\overset{n}{\underset{1}{i}}\ E_i])\ ☞\ E_1$$

$$(\text{CDR}\ [\overset{n}{\underset{1}{i}}\ E_i])\ ☞\ [\overset{n}{\underset{2}{i}}\ E_i]$$

$$(\text{CONS}\ E_1\ [\overset{n}{\underset{2}{i}}\ E_i])\ ☞\ [\overset{n}{\underset{1}{i}}\ E_i]$$

$$(\text{REVERSE}\ [\overset{n}{\underset{1}{i}}\ E_i])\ ☞\ [\overset{1}{\underset{n}{i}}\ E_i]$$

A more complex example is:

$$[\overset{n}{\underset{1}{i}}\ [\overset{i-1}{\underset{1}{j}}\ L_j\ (\text{CONS A}\ L_i)\ \overset{n}{\underset{i+1}{k}}\ L_k]]\ ☞$$

$$[[(\text{CONS A}\ L_1)\ \ldots\ L_n]\ [L_1\ (\text{CONS A}\ L_2)\ \ldots\ L_n]\ \ldots\ [L_1\ \ldots\ (\text{CONS A}\ L_n)]]$$

2) recursive

It is used to express the recursive computation performed by the function. It has the general form:

```
n_______
|       |
i  αi  -  βi  ∂        =df     α1 ... αn ∂ βn ... β1
|       |_______|
1
```

where α_i, β_i, and ∂ are parts of LISP expressions as well as linear or recursive forms. The recursive notation is essentially an indexed context-free grammar rule. It expresses *n* levels of nesting of function calls terminating with the expression ∂. When the RAINBOW system uses the recursive notation, *n* is the length of the list which is the value of the recursion variable. The crossing of lines in the notation denotes a self-reference to the entire expression with the index variable progressing one step towards the highest index.

As an example RAINBOW translates interactively, the following function definition

```
(DE append (X Y)
    (IF (NULL X) Y
        (CONS (CAR X)
              (append (CDR X) Y))))
```

into the recursive form

```
n__________
|          |
i (CONS Xi -) Y
|          |__|
1
```

which schematizes the nested expression

(CONS X_1 (CONS X_2 ... (CONS X_n Y) ...))

where X_i ***translates*** (CAR (CDR^{i-1} X))

```
 n
 |
[j Xj]   translates   X
 |
 i

 n
 |
[j Xj]   translates   (CDR X)
 |
 i+1
```

It happens that the goal of the append function is itself expressed into the reduction rule RC2:

```
n__________         m                  n     m
|          |        |                  |     |
i (CONS Xi - )  [j Yj]       ☞       [i Xi  j Yj]
|          |____||                     |     |
1                1                     1     1
```

3. USING RAINBOW: A SIMPLE REDUCTION

RAINBOW can be used interactively for the automatic documentation of programs as soon as they are typed, as advocated by (WINOGRAD 1979). In the following example, RAINBOW is reducing to its goal a set of two simple recursive functions.

The user types the following definition:

```
(DE f (X Y)
    (IF (NULL X) NIL
        (APPEND (g (CAR X) Y)
                (f (CDR X) Y))))
```

that RAINBOW translates immediately into

```
n___________________
|                   |
i (APPEND (g Xi Y) -) NIL
|                   |__|
1
```

then the following definition

```
(DE g (A Y)
    (IF (NULL Y) NIL
        (CONS (LIST A (CAR Y))
              (g A (CDR Y)))))
```

is translated into

```
n_______________
|               |
i (CONS [A Yi] -) NIL
|               |__|
1
```

then the user types to RAINBOW the symbolic call

```
(f (*list* A) (*list* B))
```

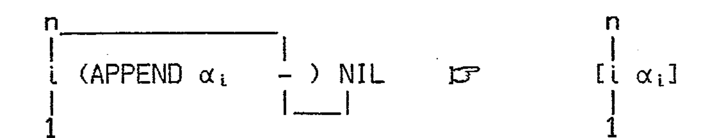
which is reduced into the linear form

```
 n m
 | |
[i j [Ai Bj]]
 | |
 1 1
```

which yields the goal of the function ***f***:
f builds the ***cartesian product*** of its two argument-lists.

The reduction set of rules which has been used in the previous example is:

- the definition of the function *g* itself.
- the rule RC1:

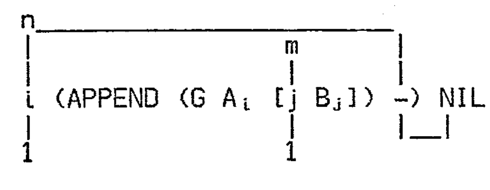


- the rule RA1:

So the RAINBOW system yields in succession :

? (f (*list* A) (*list* B))

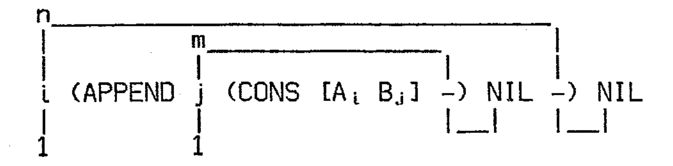
Lines beginning with "?" are typed directly by the user



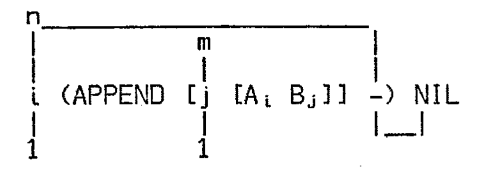
OK

? ap all

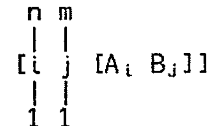
APPLYING... G GIVING...



APPLYING... RC1 GIVING...



APPLYING... RA1 GIVING...



The final linear form

4. UNDERSTANDING COMPLEX RECURSIONS

Sometimes recursion can be quite involved, as in the following square matrix generator:

```
(DE vecmat (E)
    (IF (NULL E) NIL
        (CONS E
              (DCONS (CAR E) (vecmat (CDR E))))))
```

where

```
                 n                     n
                 |                     |
(DCONS α [i Eᵢ])         ☞           [i (CONS α Eᵢ)]
                 |                     |
                 1                     1
```

The goal of VECMAT extracted by RAINBOW is:

```
          n               n  n
          |               |  |
(VECMAT [i Eᵢ])     ☞    [i [i Eᵢ]]
          |               |  |
          1               1  1
```

The body of VECMAT is slightly generalized to obtain the main reduction rule:

```
                                              n____________________________
                                              |        i-1_________       |
n________________________                     |         |          |      |
|                        |                    |         |          |      |
i (CONS αᵢ (DCONS Eᵢ     -  )) NIL    ☞       i (CONS   j (CONS Eⱼ -  ) αᵢ -  ) NIL
|                        |____|               |         |          |___|  |___|
1                                             1         1
```

which by the rule RC1 is reduced itself to:

```
                                n i-1________
                                | |          |
                        ☞      (i j (CONS Eⱼ -  ) αᵢ      (rule RDC4)
                                | |          |___|
                                1 1
```

```
                         n
                         |
Now if we sets αᵢ to   [k Eₖ], RAINBOW obtains the linear form for VECMAT by:
                         |
                         i
```

```
n i-1_________       n              n   i-1   n               n   n
| |           |      |              |   |     |               |   |
[i j (CONS Eⱼ -  )  [k Eₖ]]   ☞    [i  [j Eⱼ  k Eₖ]]    ☞    [i  [j Eⱼ]]
| |           |___||                |   |     |               |   |
1 1                i                1   1     i               1   1
```

The final result being obtained by the reduction rule RIND2:

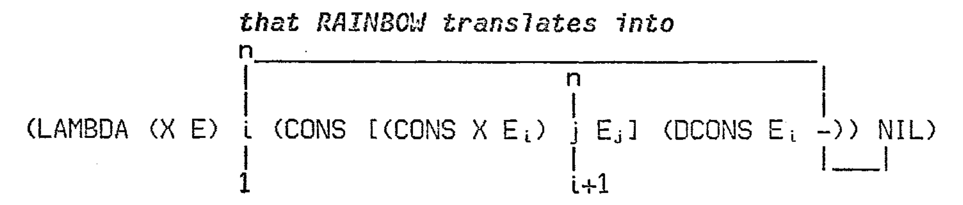
```
i-1    n                   n
|      |                   |
[i Xᵢ  j Xⱼ]      ☞       [i Xᵢ]
|      |                   |
1      i                   1
```

The same reduction process can be used to obtain the goal of the function *partinsert*, main component of a recursive partition generator. Though very usual this style of recursion is rather difficult to visualize, and is very error-prone. The following example displays a session with RAINBOW:

The user types the definition

```
? (DE partinsert (X E)
      (IF (NULL E) NIL
          (CONS (CONS (CONS X (CAR E)) (CDR E))
                (DCONS (CAR E) (partinsert X (CDR E))))))
```

that RAINBOW translates into



Then the user provides the symbolic call

```
? cl (partinsert A (*list* L))

n______________________________________
|                    n                |
|                    |                |
i (CONS [(CONS A Li) j Lj] (DCONS Li -)) NIL
|                    |                |___|
1                    i+1
```

Then the user asks for all reduction rules to be applied

```
? ap all
  APPLYING... RDC4 GIVING...

  n______________________________________________
  |       i-1________                  n        |
  |        |         |                 |        |
  i (CONS  j (CONS Lj -) [(CONS A Li)  k Lk]    -) NIL
  |        |         |__|              |        |__|
  1        1                           i+1

  APPLYING... RC1 GIVING...

   n i-1________             n
   | |          |            |
  [i j (CONS Lj -) [(CONS A Li) k Lk]]
   | |          |__|         |
   1 1                       i+1

  APPLYING... RC2 GIVING...

   n  i-1              n
   |   |               |
  [i  [j Lj (CONS A Li) k Lk]]
   |   |               |
   1   1               i+1

  OK
```

The final reduced form is the goal of partinsert

5. CONCLUSION

The 2-dimensional display for formula within RAINBOW is claimed to be intuitively understandable by using intrinsic properties of sequences: the use of indices relates the length and order of the sequence to the general form of the generic elements.

Along with the powerful reductions from recursive to linear forms, we believe that the external notations presented here reflect our intuitive understanding of lists and their properties.

In a fast checking situation, this graphical style of verification, displaying the generic structure of data, gives the user excellent control over the goal obtained from a functions sets.

The extendability provided by the incorporation of every new function definition as a new reduction rule in the spirit of (BOYER 1977) gives a useful tool to system design and programming methodology: as advocated by (GERHART 1975), the resulting schema and transformations that are saved will have to be ultimately organized into "handbooks of knowledge" about programming.

Presently, RAINBOW is able to reason about classes of data structures as sequences, we are considering its extension to arrays using the rules given in (REYNOLDS 1979). Program understanding with RAINBOW can be viewed as a kind of simplification, and its extension to several classes of data-structures may involve combination of decision procedures for several theories described in (NELSON 1978),(OPPEN 1978).

Internally, RAINBOW is mainly driven, out of the fixed initial set, by user-provided rules obtained by reduction of previous definitions to their goals. Thus, the power of RAINBOW is strictly limited by the class of expressions that the external notations are able to denote. Most of the rules in the fixed initial set are properties of the function CONS, extended to APPEND and REVERSE.

In the present state of the rules, RAINBOW is restricted to primitive recursive functions. A single recursion variable is handled within each rule : an obvious extension to an arbitrary number of recursion variables can be incorporated, each having the same pattern of sequence as an argument.

Another extension is currently implemented to handle iterative function schema as in:

```
(DE itrev (X Y)
    (IF (NULL X) Y
        (itrev (CDR X) (CONS (CAR X) Y))))
```

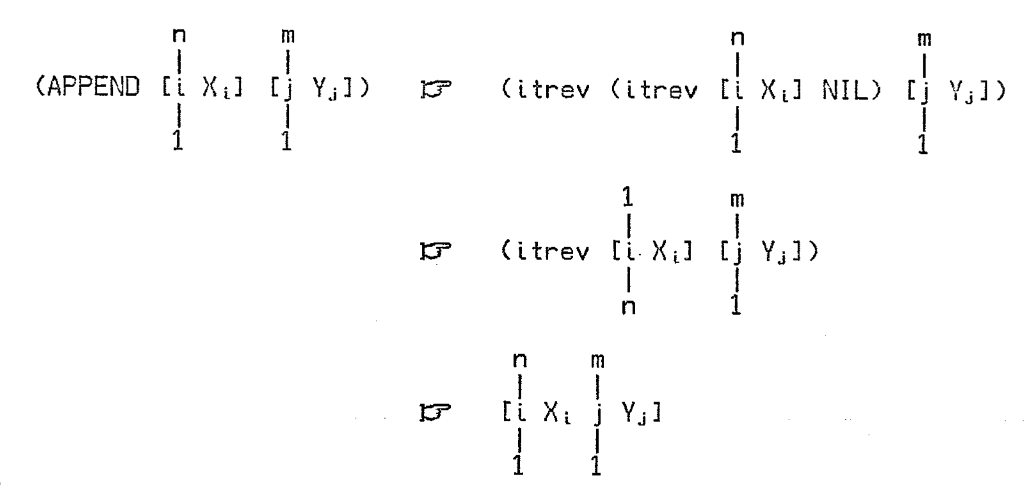
where Y is acting as an accumulator. With such an extension, the translation of *itrev* into the recursive notation should be:

```
1__________
|         |
i (CONS Xi - ) Y
|         |___|
n
```

With this extension, checking of equivalences as:

(APPEND X Y) = (itrev (itrev X NIL) Y)

could be done by direct reduction as in:



5. REFERENCES

[1] BOYER R. & MOORE J. *A Lemma Driven Automatic Theorem Prover for Recursive Function Theory*, 5th I.J.C.A.I., August 1977, Cambridge, 511-519

[2] CHAILLOUX J. *VLISP 10.3, Manuel de Reference* Universite Paris-8-Vincennes, August 1978

[3] GERHART S. L. *Knowledge about Programs: A Model and Case Study*, Proc. International Conf. on Reliable Software, 21-21 April 1975, Los Angeles, 88-95

[4] GOOSSENS D. *Meta-interpretation of Recursive List-processing Programs* I.J.C.A.I 1979, August 20-23, Tokyo, s7-s11

[5] NELSON G. & OPPEN D. C. *Simplification by Cooperating Decision Procedures*, Stanford Artificial Intelligence Laboratory, Memo AIM-311, April 1978

[6] OPPEN D. C. *Reasoning about Recursively Defined Data Structures*, 5th Annual ACM Symposium on Principles of Programming Languages, January 1978, 151-157

[7] REYNOLDS J. C. *Reasoning about Arrays*, Comm. A.C.M., May 1979, Vol 22, no 5, 290-299

[8] RICH C. & SHROBE H. E. *Initial Report on a LISP Programmer's Apprentice* IEEE Transactions on Software Engineering, SE-4:6 (1978), 456-467

[9] SHROBE H. E. & WATERS R. C. *A Hypothetical Monologue Illustrating the Knowledge Underlying Program Analysis*, M.I.T. Artificial Intelligence Laboratory, Memo 507, January 1979

[10] WERTZ H. *Automatic Program Debugging* I.J.C.A.I 1979, August 20-23, Tokyo, 951-953

[11] WINOGRAD T., *Beyond Programming Languages* C.A.C.M., Vol. 22, no. 7, July 1979, 391-401

Stereotyped Program Debugging : an Aid for Novice Programmers

Harald WERTZ

C.N.R.S. LA 248 L.I.T.P.
2 place Jussieu
75005 Paris
&
D¬partement d'Informatique
Universit¬ Paris VIII - Vincennes
75571 PARIS C¬dex 12

Summary : This paper presents a system (PHENARETE) which understands and improves incompletely defined LISP programs, such as those written by students beginning to program in LISP. This system takes, as input, the program without any additional information. In order to understand the program, the system meta-evaluates it, using a library of ***pragmatic rules***, describing the construction and correction of general program constructs, and a set of ***specialists***, describing the syntax and semantics of the standard LISP functions. The system can use its understanding of the program to detect errors in it, to eliminate them and, eventually, to justify its proposed modifications. This paper gives a brief survey of the working of the system, emphasizing some commented examples.

Introduction

Much effort is spent on the development of tools to help programmers in constructing, debugging and verifying programs. From simple editors and trace-packages, the trend is towards more and more sophisticated automatic programmers [3,8], auotomatic debuggers [15,17,1] automatic assistants [14], automatic verifiers [11,5] or even the construction of new - semantically more firmly based - programming languages (PASCAL, ALPHARD).

Most of these tools exhibit some weaknesses such as

- they impose too many constraints on the intuitions of the programmer (cf [6]),

- they work only on a very limited subset of possible programs (cf [1,15]),

- they work only on correct programs (cf [2,11]).

Our aim in the design of our program understanding system was four-fold :

-1- we wanted to have a system which makes explicit the knowledge involved in constructing and debugging programs;

-2- we wanted our system not to verify the correctness of programs but their *consistency*

-3- we wanted the system to provide *hints* for improving and correcting programs and

-4- we wanted a practical, useful and running system.

Thereupon we have built a program understanding system able to automatically correct and improve programs. This system, PHENARETE, *assists* beginning programmers during the writing and debugging of their programs.

Programming errors

The construction of a computer program can be divided into different steps : first, the programmer has to have a somewhat precise conceptualisation of the activity he disires the computer to perform. As quoted by Model [13], this conceptualisation doesn't just restrain at the level of overt input/output behaviour, but includes the projected programs manipulation of its internal data structures. Second, the programmer has to conceive a method by which the intended activity may be performed. Third, he has to express this method in terms of a programming language and, fourth, he has to communicate the program to the computer.

This process induces 5 major types of programming errors, related to these different steps : lexical, syntactic, semantic, teleological and conceptual errors.

Lexical errors are probably the most frequent - and more easily debuggable - errors. They are mainly misspellings or typographical mistakes, and are detectable at the word-level. They refer - in classical compilers - to the lexical analyser or scanner.

Every language has rules governing the allowed forms of program statements. We call deviations of these rules syntactic errors.

These first two kinds of errors, normally detected during the reading of the program (compilation phase), are - in this paper - invariably called surface errors or informalities.

Another kind of errors is that - even when the syntactic form is correct - its meaning may be unclear, contradictory or invalid. Examples of this kind of errors are "division by zero", an attempt to multiply a number by a string or a call of a function which has no definition. We call such errors semantic errors. They are normally detected during the run-time of the program.

New algorithmic languages, such as PASCAL, MESA or ADA, try to detect semantic errors at the semantic level through "strong compile-time checking of data types and program interfaces". But even when they are detected by syntactic checking, the sources of such errors lie at the semantic level.
Let us note that such compile-time checking is rather limited, since even such a simple error as division by zero escapes the checking if the divisor isn't just a constant or a variable, but an expression whose value

is computed at run-time and the expression equals zero only sometimes.

The fourth class of errors we can distinguish, are teleological errors, which occur when the program does something, but not what was intended. These errors refer to problems in the scope of the programmers' precise specification of the computational scheme or algorithm.

In our paper we call errors of the last two kinds deep errors or inconsistencies.

A last kind of errors is met if the source of the problem is in the method or approach devised by the programmer to achieve the desired computational activity. These errors are called conceptual errors.

Our system, PHENARETE, analyses the text of a program and, when it detects informalities or inconsistencies, it constructs and proposes possible corrections or improvements. These modified versions of the program, constitute for the novice programmer, a crucial component of his apprenticeship : basing on *examples* the process of improving and correcting imperfect programs, and basing on *models* the process of inventing future programs.

The originality of the system resides 1) in its ability not only to detect low-order errors - a standard in todays compilers - but also middle- and even higher order errors (i.e. semantic and some teleological errors), and 2) in its attempt to use the knowledge necessary to detect errors, also to construct possible corrections.

Overview of the system

The system takes as input the draft version of a LISP program and delivers as result of its treatment one or more versions of the same program, corrected and improved. PHENARETE proceeds in several steps (cf fig. 1) : first a preliminary analysis of the text of the program is performed in order to detect and correct surface errors. By surface errors we mean errors detectable by local anlysis : simple lexical errors such as misspellings or parenthesis errors as well as syntactic ones such as errors concerning the wrong number of arguments in function calls. During this first analysis PHENARETE collects information for the following steps. Whenever it meets the name of a function (standard or user defined LISP function) it automatically activates a *specialist*, associated with this function, by the process of data driven function invocation [16]. These specialists represent its knowledge about the use and the effects of the associated functions.

figure 1

The result of the first analysis - a syntactically correct program - is then meta-evaluated to verify the programs *well-formedness*. We say that a program is well-formed if it doesn't contain obvious infinite loops, doesn't have useless statements (i.e. statements never executed) and if it doesn't contradict the set of rules incorporated in PHENARETE. Every final version of the program our system delivers is a well-formed program.

When PHENARETE detects some informalities or inconsistencies, it annotates the corresponding part of the program and sends the code and the annotation to the repair-module, a module designed to eliminate errors (with the help of stereotyped programming knowledge). The system stops the analysis process when not finding any more possible improvements.

Organisation of PHENARETE

The system is based on four main concepts :

-1- During the analysis of a program, PHENARETE constructs a description - an internal representation - of the program in terms of ***cognitive atoms***. These may be considered as the nodes of a network-like representation of the program.

Each cognitive atom is composed of a set of facets which represent its different aspects, about which questions can be asked, such as

- what type is it ?
- what is its definition ?
- (if it is a function) what is its domain ?
...

The facets are filled in during the analysis process.

We distinguish between three classes of cognitive atoms, those concerning variables, those concerning labels and those concerning functions. The facets are the same for every atom inside the same class, but they differ significantly from one class to another.

The representation of a cognitive atom is a set of attribute/value pairs, and the value V found under the attribute P of atom A is simply the answer the expert A can give to question P.

-2- A set of ***specialists***, i.e. a set of procedural specifications of the syntax and the operational semantics of the standard LISP functions, such as CAR, CDR, EQ, COND etc.

We distinguish between two different types of specialists :

- those describing the syntax and

- those describing the semantics

of the associated functions.

example : specialist CAR for the syntax

```
[CAR-1 (X) =>
        v (& atom (CAR X)
           & type (X) = LISTP)
        v (& S-expression (CAR X)
           & type (val (X)) = LISTP)
  else :
           modify X until CAR-1 (X) = T]
```

paraphrasing :

CAR expects that its argument is
- an atom
  and the type of the value of the argument is a list
- a S-expression
  and the type of the value of that S-expression is a list (e.g. a function call)

else
  CAR has to modify the argument until one of these two conditions is true

and the specialist CAR for the semantics

```
[CAR-N =>
    arg : (X (meta-eval X))
   test : (type (val (X)) = LISTP) ->
          (type (val (X)) = ?)
            -> hypothesize (X, type: LISTP)
          T -> complain (X, type : LISTP)
 action : if (exist (CAR X)) --> (CAR X)
         else (create (CAR, X)) --> (CAR X)]
```

or in paraphrasing :

CAR-N

has an argument named X, which must be evaluated
one must verify
  if
    the type of value of the argument is a list, all is ok
  else
    if the type of value of the argument isn't known, one has to create a hypothetical value of type LIST for X
  else
    one has to call the repair-module to change the text of the program in such a way that the value of X becomes a list

the value of CAR is
  if
    there exists already a CAR of X, this CAR
  else
    one has to create a symbolic value for X, the CAR of which
    will be the desired value

Note that the internal description constructed during the analysis of each user function is used to construct two new such specialists for each user function.

The specialists are the agents of the meta-evaluation and they represent the systems knowledge about the programming language used.

-3- An algorithm of ***meta-evaluation*** [7] which helps the system to analyze each of the possible paths of the program. This algorithm is composed of two parts :

- a module to determine the symbolic values, i.e. since the meta-evaluation doesn't use concrete values in order to evaluate the program, it must deduce from the text of the program the type and the structure of the arguments of the different functions to evaluate. Thus our algorithm is rather different from those proposed in [10, 12] which receive the symbolic values as input.

- an evaluation module, i.e. a module able to evaluate functions on these symbolic arguments.

With this algorithm one ***unique*** execution is representative for every particular execution [12].

-4- A set of ***pragmatic rules*** describing general program constructs and stereotyped methods to repair inconsistencies. These rules formalize and express explicitly the knowledge activated by every programmer when he is reading a program.

We do not use rules concerning the task domain of the programs : our intention was to build a system which does not ask for any additional information, i.e. which works only with the text of the program to be analyzed, so that the programmer is not obliged to accompany the programs by comments, assertions or other specifications. The system should work - without any additional information - in any possible task domain, numeric as well as symbolic.

Our rules are very general and valid for every LISP program obeying to the following restrictions :

- the names of variables, functions and labels are unique

- each function call must be of the type "call by value"

- the unique functional arguments permitted are explicit λ-expressions.

We call this subset of LISP : "extended first order LISP".

examples of pragmatic rules :

(1)

```
rule of the dependence of a loop of the
predicate =>

 given :  F ∈ {loop}
          T = {exit-test of F}
          Y = {variable} so that
                              ∀y∈Y ¬t∈T y⊂t
 then :   ∀y∈Y val(y) [F]1 val'(y)
              ∧ val(y) = val'(y)
                 => val(F) = undefined
```

in paraphrasing :

> if no variable of the exit-test of a loop is modified inside the loop-body, then the loop is independent of the exit-test and either its execution is non-terminating or the loop will never be executed.

A slight refinement of this rule is the following one :

(2)

rule of the structural well-formedness of loops =>

> in a loop at least one of the variables used in the exit-test has to change its value inside the body of the loop, in such a way that its structure is simplified and converges towards the satisfaction of the exit-test.

In order to use this rule, we have implemented a small theorem prover which can prove the convergence by induction. As to the proof implied by this rule, the theorem-prover takes the symbolic value of the variables at the entry of the loop, and the modified symbolic values of the same variables after *one* meta-evaluation of the loop-body and tries to prove the convergence towards the stop test.

Let us give one last example of a pragmatic rule :

(3)

```
rule of the structure of recursive loops =>

given : F ∈ {recursive function}
        A = {call of F} so that A⊂F
        S = {selection-clause} so that S⊂F
then :  ∀a∈A ¬s∈S a⊂s
          ∧ ¬s∈S so that ∀a∈A a⊂s
```

paraphrasing :

in a recursive function, the recursive calls have to be inside a selection-clause, and at least one of the clauses must not contain a recursive call.

Presently we have about a hundred rules incorporated in the system, dealing especially with iteration and recursion, the use of variables and list-processing. For almost every rule there exists a dual one indicating how to modify the program in such a way that the first one is satisfied.

Some commented examples

To use PHENARETE, the user has to give the system only the text of the draft version of the program he wants to write, without any additional information like input/output assertions, comments, plans etc. The system will try to understand what the user wants to do and, if necessary, modify the text of the programm.

To give some feeling how the system works, let us examine some examples in detail :

Our first example is a (very) erroneous version of the well known REVERSE function. Here is the actual input to the system :

```
? (P '(DE REV L1 L2 COND ULL L22 A1 T RVE A1 ONS CRA A1 A2))
```

PHENARETE will first perform a preliminary analysis using only its syntactic knowledge, i.e. the specialists for the syntax and the spelling error corrector. The result of this first analysis is a syntactically correct LISP progam, a program accepted by any LISP interpreter or compiler.

Here is the actual printout of PHENARETE at this point :

```
ERROR:
          NAME --> (? ULL --> NULL)
ERROR:
          NAME --> (? L22 --> L2)
ERROR:
          NAME --> (? A1 --> L1)
ERROR:
          NAME --> (? A1 --> L1)
ERROR:
          NAME --> (? RVE --> REV)
ERROR:
          NAME --> (? A1 --> L1)
ERROR:
          NAME --> (? ONS --> CONS)
ERROR:
          NAME --> (? CRA --> CAR)
ERROR:
          NAME --> (? A1 --> L1)
ERROR:
          NAME --> (? A2 --> L2)
```

SURFACE IMPROVEMENTS :

```
(DE REV (L1 L2)
   (COND
      ((NULL L2) L1)
      (T (REV L1 (CONS (CAR L1) L2)))))
```

These first improvements have eliminated all the syntactic errors. However, at least one semantic error remains :

> Neither L1 nor L2 are modified in such a way that their values converge towards the stop test; even with a modification of L1 in the recursive call, the recursion won't stop since the stop-test has as argument L2, a list which grows longer and longer on successive recursive calls (rule 2).

During every analysis, our system collects useful information about the program, represented in the data-base which constitutes the internal description of the analyzed program. Figure 2 shows the state of the data-base after the first analysis of the function REV.

----- figure 2 somewhere here ----

data-base after the first analysis

figure 2

The knowledge represented in fig. 2 can be paraphrased as follows :

> After the first analysis PHENARETE knows that REV is a recursive function, with two arguments. The first argument is called L1, which the system doesn't know the value of, but knows it has to be a list. At the recursive call of REV, the first argument is not modified.
>
> The second argument is called L2. PHENARETE doesn't know its value either. It too has to be a list. At the recursive call L2 takes the value of the expression (CONS (CAR L1) L2).

As for the errors detected up to that point, most standard scanners detect this kind of errors, and some compilers even handle this kind of corrections (PL/C or CORC).

For the further improvements, the system proceeds to a meta-evaluation of the current version, using as symbolic value of the arguments just the knowledge that they have to be lists. If further specification about the values is needed, PHENARETE constructs them dynamically.

In this example, the system has - corresponding to the two clauses of the selection (COND) - to develop two paths, one supposing the predicate (NULL L2) satisfied, one supposing it falsified.

Note that one of the main differences between standard evaluation and

meta-evaluation is that the knowledge derived during the meta-evaluation of one of the possible paths can be used during the meta-evaluation of every other path.

After one meta-evaluation of the function REV, PHENARETE has constructed an evaluation tree of the form :

```
              o REV list-L1 list-L2
              |
              o COND
             /
     L2=() /     L1=(a1 ... an)
          /      L2=(b1 ... bn)
         /
        o        o
    result=L1    REV (a1 ... an)
                     (CONS a1 (b1 ... bn))
```

As noted above, the meta-evaluation stops, when it reaches a point where no further continuation is possible (the case of the left branch) or when it reaches a point where control has already passed by (which will be the case later on, during the meta-evaluation of the right branch of the tree).

When the system meets a recursive call or an iteration, it carries on the evaluation just as in normal execution. Here the continuation of the right branch of the tree forces the system to falsifie the first predicate of REV, since the symbolic value of L2 is a CONS-expression which can't equal NIL (the empty list). So we find :

```
              o REV list-L1 list-L2
              |
              o COND
             /
   L2=()    /    L1=(a1 ... an)
           /     L2=(b1 ... bn)
          /
        o        o
   result=L1   REV (a1 ... an)
                   (CONS a1 (b1 ... b2))
                 |
                 o COND
                 |
                 | L1=(a1 ... an)
                 | L2=(CONS a1
                 |       (b1 ... bn))
                 o
               REV (a1 ... an)
                   (CONS a1
                     (CONS a1
                       (b1 ... bn)))
```

At this point PHENARETE has to stop the meta-evaluation : all terminal leaves are either end-points of the program or points already visited during the meta-evaluation.

Application of the pragmatic rules, especially those concerning recursion, indicates that the program won't stop, since the modification of the arguments isn't in such a way that they converge towards satisfaction of the exit-test (rule 2) : the exit-test checks if L2 is a empty list, but L2 grows longer during the successive recursive calls. The demonstration here is straightforward, but we easily realize that the power of the system depends crucially on the power of the incorporated theorem prover.

Each rule is combined with an ordered set of *advices* as to "how to correct" the non-satisfaction of the rule. The advices of rule 2 say that in order to correct the code, we have to suppose that

-1- there is an error in the stop test or

-2- there is a missing stop test or

-3- there is an error in the recursive call

The pragmatic rules as well as the advices (which are just pragmatic rules of correction) are hierarchically organized, from the general to the concrete, i.e. the application of one rule may result in the application of a lot of other rules.

In this case the first advice PHENARETE finds applicable is the third one. Here is another point of the system which deserves some critics : PHENARETE supposes that at least one of the arguments has to simplify its structure in the successive calls. If this isn't the case, and the system doesn't find an error, all is ok, but otherwise it forces a simplification. Here the simplification is forced by the use of the CAR of L1 : PHENARETE assumes the user wants to work with successive elements of L1, which makes it introduce the modified recursive call

```
(REV (CDR L)(CONS (CAR L1) L2))
```

No doubt, if the programmer didn't want that, eventually he won't understand any more the correction of PHENARETE. But, in constructing the system we wanted the programmer not to be obliged to give asssertions or comments, so we have no means to know anything about the intentions of the programmer (except what is explicitly stated in the code), and the proposed corrections may differ significantly of those used by an experienced programmer knowing the intentions.

But, to justify our approach, before beginning to write the system, we have systematically studied, during one year, the programs written by our students. The corrections PHENARETE proposes, are those corresponding to the statistically most current errors. On the other hand, we think a program which stops and delivers a result different from the intended one is easier to debug than a program entering in an infinite loop. PHENARETE just corrects progams in such a way that they don't abort and stop delivering a result, when executed.

After each modification, the system re-meta-evaluates the program, to check if any error remains.

In our example PHENARETE can not disambiguate this function - it does not know anything of the intentions of the programmer - so it gives two

different propositions :

PROPOSITION 1 :

```
(DE REV (L1 L2)
   (COND
      ((NULL L2) L1)
      ((NULL L1) L2)
      (T (REV (CDR L1) (CONS (CAR L1) L2)))))

            AT LEAST YOUR FUNCTION SEEMS OK.
```

In this first proposition, PHENARETE supposed the given stop-test correct, but assumed that the user omitted a second stop-test in case the second argument is not NULL at the initial call of REV.

PROPOSITION 2 :

```
(DE REV (L1 L2)
   (COND
      ((NULL L1) L2)
      (T (REV (CDR L1) (CONS (CAR L1) L2)))))

            AT LEAST YOUR FUNCTION SEEMS OK.
```

In this second proposition, PHENARETE supposed that the user inadvertently inverted the arguments of the stop-test, so it inverts the two arguments L1 and L2.

PHENARETE secures that the two corrected versions of the initial draft-program (which are not identical) will stop and deliver a result when executed.

The data-base after the completion of the analysis of REV is represented in figure 3 (only for the second version) :

figure 3 here somewhere

data-base after the analysis

figure 3

Two sources of information are available after the complete analysis of a program :

- the program as an executable algorithm

- the description of the program in the data-base

This data-base doesn't describe 'how' or 'what' the program computes, but gives valuable indications on the conditions which have to be verified if the program is activated and on those which are verified after its

execution.

This description is used to construct new specialists, in order not to reanalyze the function if it is called by another one.

Here are the two specialists :

specialist REV for the syntax

```
[REV-1 (X) =>
        & (v (& atom (CAR X)
               & type (CAR X) = LISTP)
           v (& S-expression (CAR X)
               & type (val (CAR X)) = LISTP))
        & (v (& atom (CADR X)
               (& type (CADR X) = LISTP)
           v (& S-expression (CADR X)
               & type (val (CADR X)) = LISTP
    else :
      modify X until REV-1 (X) = T
```

specialist REV for the semantics

```
[REV-N =>
  arg : (X Y (meta-eval (X) meta-eval (Y)))
 test : (type (X) = LISTP)
          -> (type (Y) = LISTP) ->
             (type (Y) = ?)
                -> hypothesize (Y, type:LISTP)
             T -> complain (Y, type:LISTP)
        (type (X) = ?)
          -> hypothesize (X, type:LISTP)
             go test
        T -> complain (X, type:LISTP)
             go test
action: => {LISTP, execute-symb (REV X Y)}]
```

As can be easily verified, the specialists are just algorithmic transcriptions of the data-base.

Now, when PHENARETE meets, during the analysis of another program, a call to the REV-function, it hasn't to reanalyze the body of the function, but it can use the specialists of REV in the same manner it uses the specialists of the standard function.

Our second example is an extremely "simplified" version of the equally well-known FACTORIAL function. Here it is :

```
? (DE FACT N TIMES N FACT N)
```

As in the previous example, PHENARETE will first translate this unparenthesized expression into an well parenthesized one :

SURFACE IMPROVEMENTS :

```
(DE FACT (N) (TIMES N (FACT N)))
```

This first proposition is a syntactically correct program, but semantically it presents some difficulties :

-1 at the recursive call, N is not modified (rule 1).

-2 there is no stop-test at all (rule 3), so there are two (!) reasons to make the recursion infinite.

Remember that PHENARETE doesn't know the intentions of the programmer, so it must detect these errors without any additional information : all that can be used in the further analysis are the semantic specialists and the pragmatic rules. So let us look at its versions :

P R O P O S I T I O N :

```
(DE FACT (N) (COND
   ((LE N 0) 1)
   (T (TIMES N (FACT (SUB1 N))))))

         AT LEAST YOUR FUNCTION SEEMS OK.
```

This corrected version is actually a correct version of the factorial-program. The performance illustrates well the pertinance of the pragmatic rules, knowing that the system works completely automatically without asking any questions to the user and without any information about the supposed intention.

One last example :

```
? (DE ADDIT M N  (((ZEROP N) M))
                 (T (ADDIT SUB1 M ADD1 N)))
```

SURFACE IMPROVEMENTS :

```
(DE ADDIT (M N)
    (COND
       ((ZEROP N) M)
       (T (ADDIT (SUB1 M) (ADD1 N)))))
```

P R O P O S I T I O N :

```
(DE ADDIT (M N)
    (COND
       ((ZEROP N) M)
```

```
((LE M O) N)
(T (ADDIT (SUB1 M) (ADD1 N)))))

     AT LEAST YOUR FUNCTION SEEMS OK.
```

The difference between this example and the previous one consists mainly in the constructed return value of the function. In the FACT example, the value returned was just the neutral element of the operation applied to the result of FACT (e.g. TIMES), in this example a value is computed during the recursion, thus this newly computed value will be the result of the function.

Conclusion

We have presented a running system which corrects incorrect student programs in some stereotyped way.

There are two major shortcomings we would like to mention :

-1- Our system relies heavily upon a (still very small) theorem-prover, and - related to it - the internal representation of the abstract data-structures. These two modules are currently much too limited to apply the system to real user progams (and not only to small student programs). But looking at most other systems of meta-evaluation [10, 12 for example] it is one of the first systems carrying on the meta-evaluation not only on numeric programs but on symbolic ones, where no algebraic theory yet exists.

We think we will use for the next version of the system the abstract list-representation proposed by Goossens [7].

-2- The total absence of any mean to communicate to the system the intentions of the programmer constitutes - as soon as one leaves more or less toy programs - a severe handicap. Presently we are investigating possibilities to encompass this shortcoming. Different approaches are possible and we don't know yet which one to choose : either allow comments (but what should we do if there are errors in the comments?) or oblige the programmers to give some input/output examples, which would permit to use real *and* symbolic values (but same question as above).

We are presently implementing PHENARETE as a standard VLISP [4, 9] error routine, i.e. as soon as during the execution of a program an error occurs, we trap it and apply the system to the program which caused the error. This will eliminate the - always existing - possibility that the program modifies a correct - but for it unintelligible - program.

A word about LISP : the only reason why we have choosen LISP is the isomorphism between external and internal representations, and because our students learn LISP as first programming language. The algorithms and rules used are very general and not LISP-specific. An implementation of PHENARETE for an algorithmic language, say Pascal for example, seems straightforward : the system would even have less work in determinig the

symbolic values because of the typed declaration of all identifiers.

The system is running on PDP-10, uses about 25k word memory, is implemented in VLISP [4,9], and is used by about 600 students in our university.

A more detailed description may be found in [18].

REFERENCES

[1] ADAM A. & LAURENT J.P. *Transformation de Programmes et Correction de Programmes.* Colloque sur l'Intelligence Artificielle, Strasbourg, CNRS, (1977), pp 41-83

[2] ARSAC J. *La construction de programmes structurés.* Dunod-Informatique, Paris, 1977

[3] BALZER R. *Automatic Programming.* (Draft), ISI, University of Southern California, 1972

[4] CHAILLOUX J. *VLISP-10.3 manuel de référence.* D¬pt. Informatique, Universit¬ Paris 8, RT-17-78, 1978

[5] DEUTSCH P.L. *An Interactive Program Verifier.* Xerox Parc, CSL 73-1, 1973

[6] DIJKSTRA E.W. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976

[7] GOOSSENS D. *A System For Visual-Like Understanding of LISP Programs.* Proc. AISB/GI Conference, Hamburg, RFA, july 17-19, 1978

[8] GREEN C. & BARSTOW D. *On Program Synthesis Knowledge.* Artif. Intell. 10:3 (1978) 241-279

[9] GREUSSAY P. *Contribution a la Définition Interprétative et a l'Implémentation des Lambda-langages.* These, Universit¬ Paris 7, 1977

[10] HOWDEN W.E. *Symbolic Testing and the DISSECT Symbolic Evaluation System.* IEEE Transaction on Software Engineering SE-3:4 (1977) 266-278

[11] IGARASHI S., LONDON R.L. & LUCKHAM D.C. *Automatic Program Verification 1 : Logical Basis and its Implementation.* Acta Informatica, vol. 4, (1975) 145-182.

[12] KING J.C. *A New Approach to Program Testing.* Proc. ACM Int. Conf. on Reliable Software, Los Angeles, (1975), pp 228-233

[13] MODEL M.L. *Monitoring System Behaviour In a Complex Computational Environnement.* Xerox, Palo Alto Research Center Ca, CSL-79-1, 1979

[14] RICH C. & SHROBE H.E. *Initial Report on a LISP Programmer's Apprentice.* IEEE Transactions on Software Engineering SE-4:6 (1978) 456-467

[14] RUTH G.R. *Analysis of Algorithm Implementations.* M.I.T., MAC-TR-130, 1974

[16] SANDEWALL E. *Ideas About Management of LISP Data Bases.* AI-Memo-332, M.I.T., Cambridge, 1975

[17] TEITELMAN W. *INTERLISP Reference Manual.* Xerox Parc, Palo Alto, 1974

[18] WERTZ H. *Un système de compréhension, d'amélioration et de correction de programmes incorrects.* Thεse de 3εme cycle, Universit¬ Paris 6, 1978

LA META-EVALUATION AU SERVICE DE LA COMPREHENSION DE PROGRAMMES

Daniel GOOSSENS

Le programmeur expert qui écrit, étend ou modifie en parallèle plusieurs programmes participant tous d'un système unique, aura besoin d'un assistant qui ***dénonce*** interactivement les imperfections issues de son incapacité à contrôler l'organisation globale de ce système.

C'est à étudier comment atteindre ce but qu'est consacré le système CAN, un système de compréhension automatique de programmes LISP, présenté dans cette étude. CAN est implémenté en VLISP [GREUSSAY 76 77 78 79, CHAILLOUX 78a 78b 78c] et tourne sur un PDP-10 modèle KI-10. Il y occupe 20K mots de 36 bits.
CAN a la capacité de déceler dans un système dont il ne connait pas l'intention des particularités telles que :

- portions de code non utilisées
- classes de données pour lesquelles le calcul ne termine pas
- redondances
- effets de bord indésirables

La compréhension dont CAN est capable est fondée sur un processus compact de ***méta-évaluation***. La méta-évaluation permet d'effectuer en un seul calcul symbolique une infinité de calculs particuliers. La méta-évaluation utilisée par CAN représente les connaissances au sujet de programmes sous forme de systèmes d'équations et se charge de les résoudre. Elle sert à analyser dans un programme le flot des données, dans le but d'en exhiber des particularités (calculs infinis, portions de code non utilisées, chemins impossibles, redondances). La méta-évaluation utilisée par le système CAN répond à deux types de problèmes posés par l'évaluation symbolique:

- L'évaluation symbolique, dans ses réalisations actuelles [LONDON 74, KING 75 76, BOYER 75b, YONEZAWA 75 76, BALZER 77, CHEATHAM 79] ne permet pas que soit décrit de façon symbolique autre chose que les valeurs associées aux identificateurs, dans un environnement. De ce fait, elle ne traite pas de nombreuses constructions, fréquement utilisées en LISP: variables fonctionelles, macro-génération dynamique de fonctions incomplètement spécifiées, définition de structures de contrôle.

- Les résultats fournis par l'évaluateur symbolique sont souvent aussi peu transparents que le programme évalué symboliquement, surtout en ce qui concerne les programmes de traitement de listes. Les représentations traditionellement utilisées par l'évaluation symbolique, les formules de la logique des prédicats, n'exhibent pas plus que les programmes qu'elles annotent, les redondances, les classes de données pour lesquelles le programme ne termine pas son calcul, les propriétés <u>utiles</u> de programmes.

La META-EVALUATION utilisée par le système CAN augmente l'évaluation symbolique dans ces deux directions:

- Elle l'étend aux constructions nouvelles permises par LISP: style applicatif, macro-génération de programmes, affectations où l'identificateur n'est pas explicité, structures de contrôle définies par l'utilisateur, appels à l'évaluateur, programmation incrémentale, entrées-sorties.

- Elle repose sur un nouveau mode de représentation des données symboliques: les ***représentations conceptuelles*** [YONEZAWA 76]. CAN est capable d'associer à un programme une représentation conceptuelle de son activité et de ses propriétés principales. Cette traduction en représentations conceptuelles a pour but d'isoler les problèmes de compréhension au sujet d'un complexe de programmes, sous forme d'***équations*** à résoudre. CAN utilise alors des systèmes indépendants de résolution d'équations, spécialisés en particulier dans le traitement de listes.

La méta-évaluation doit être utilisée et contrôlée par des systèmes plus larges de ***compréhension automatique*** de programmes.

Considéré comme un système de compréhension de programmes LISP, CAN associe à un programme une définition sémantique, qui lui est équivalente, mais de laquelle il est beaucoup plus aisé d'extraire des propriétés importantes. Les langages de programmation, et LISP en particulier, déploient une immense variété de représentations de calculs, et permettent d'oblitérer des propriétés simples derrière des formules courtes, mais à la sémantique complexe.
Pour CAN, comprendre un programme, c'est y associer sa représentation conceptuelle. Les représentations conceptuelles forcent l'extraction de ce qui est habituellement caché dans un programme: effets de bord, contrôle de l'évaluation, analyses de cas, structure dymamique des objets manipulés par le programme. Elle peuvent ainsi isoler des particularités interessantes d'un programme, telles que:

- effets de bord imprévus
- classes de données pour lesquelles le calcul du programme ne termine pas
- redondances
- portions de code non utilisées.

L'EVALUATION SYMBOLIQUE

Il est possible de modifier un interprète pour qu'il tienne compte de la présence de ***valeurs abstraites*** dans l'environnement [note 1] d'une expression à évaluer. Evaluer symboliquement une expression, c'est l'évaluer dans un environnement insuffisamment décrit pour pouvoir utiliser un évaluateur classique. Evaluer symboliquement cette expression consiste à compléter la description de l'environnement sur la seule base des exigences de cette expression, et puis évaluer cette

expression normalement.

LA META-EVALUATION

Le domaine d'entrée de l'évaluateur EVAL du langage LISP est l'ensemble des couples <EXPRESSION , ENVIRONNEMENT> où EXPRESSION est une expression LISP, et où ENVIRONNEMENT donne accès à une liste de couples <IDENTIFICATEUR , VALEUR>.

Le domaine d'entrée du méta-évaluateur est l'ensemble des couples *⟨a,b⟩* où *a* et *b* sont respectivement une expression et un environnement méta-décrits.

L'évaluateur symbolique admet que VALEUR soit méta-décrit, mais s'attend à ce que IDENTIFICATEUR et EXPRESSION soient des données concrètes.
l'examen du langage LISP montre qu'il est nécéssaire que l'évaluateur symbolique s'attende à ce que IDENTIFICATEUR et EXPRESSION soient méta-décrits. Il doit être étendu en un méta-évaluateur.

Exemple :

Le langage LISP autorise l'usage d'arguments fonctionnels. Comprendre un programme qui possède des arguments fonctionnels, c'est pouvoir simuler l'application sur des données d'un programme insuffisamment décrit, c'est-à-dire un schéma de programmes.

Le langage LISP permet, par exemple, qu'en position de fonction, dans une expression non atomique, se trouve une autre expression à évaluer, censée s'évaluer en une fonction qui sera alors appliquée sur les arguments.

On peut écrire en LISP des programmes qui construisent d'autres programmes à partir de données, et qui les utilisent.

L'exemple suivant, simplifié, est tiré du générateur de conditions à vérifier de [IGARASHI 73] programmé en VLISP.

```
(DE FOO (a b)
    [λ [a]
       ['SUBSTITUER
        ['QUOTE a]
        ['QUOTE b]]])

(DE SUBSTITUER (s l)
```

[note 1] Pour l'évaluateur symbolique présenté dans ce chapitre, l'environnement est une liste de couples (identifieur,valeur). Le mode de définition de l'évaluateur symbolique laisse cependant la possibilité de détailler d'autres structures dans l'environnement. C'est ce qui est fait au chapitre 4, où l'on y introduit une pile pour les fonctions d'échappement, et des tampons d'entrée et de sortie.

```
(SUBST (EVAL s) s l))
```

Dans un contexte créé par un appel de FOO, SUBSTITUER remplace dans la liste contenue dans l toutes les occurences de l'atome contenu dans s par la valeur de cet atome dans l'environnement courant.
L'erreur contenue dans cet exemple est de même nature que celle contenue dans le programme FILTRER, exemple du paragraphe 4.1.1, quoique plus complexe. Les méthodes classiques de test sur exemples précis ne marchent donc pas plus.

Le résultat d'un appel de FOO est utilisé comme un programme et appliqué sur une valeur quelconque. Voici le comportement souhaité de l'évaluateur symbolique:

<((FOO *'a* *'b)* *'v*) , {}>

commentaire: le travail de FOO n'est pas détaillé. Une fois sa valeur retournée, les liaisons de ses paramètres formels sont détruites.

<((λ (*a*) (SUBSTITUER (QUOTE *a*) (QUOTE *b*))) *'v*) , {}>

commentaire: L'évaluateur symbolique doit utiliser ici des connaissances sur la syntaxe des λ-expressions et se polariser sur le cas où ***a*** est un atome littéral.

<(SUBSTITUER (QUOTE *a*) (QUOTE *b*)) , {*a*=*v*}>

<((λ (s l) (SUBST (EVAL s) s l)) (QUOTE *a*) (QUOTE *b*)) , {*a*=*v*}>

commentaire: pour affecter les valeurs symboliques ***a*** à s et ***b*** à l, l'évaluateur symbolique tient compte des possibilités:

a=s	*a*=l	*a*≠s, *a*≠l
<(SUBST (EVAL s) s l) , {s=s l=*b*}>	<(SUBST (EVAL s) s l) , {s=l l=*b*}>	<(SUBST (EVAL s) s l) , {s=*a* l=*b* *a*=*v*}>
(SUBST 's 's '*b*)	(SUBST '*b* 's '*b*)	(SUBST '*v* '*a* '*b*)

(SUBST x y z) substitue x à y dans z. Seule la troisième branche correspond à l'intention initiale, qui est de substituer dans ***b*** toute occurence de ***a*** par ***v***.

L'évaluation symbolique fournit les exemples pour lesquels l'intention n'est pas satisfaite:

1. ((FOO 's '*b*) '*v*)

2. ((FOO 'l '*b*) '*v*)

où s et l sont des atomes litéraux et *b* et *v* des valeurs abstraites.

LA RESOLUTION D'EQUATIONS

Les définitions sémantiques des fonctions de base ne décrivent plus les raisonnements à effectuer devant tous les types de situations rencontrables. Ces raisonnements sont isolés en systèmes indépendants. Les définitions sémantiques se contentent de décrire statiquement le domaine d'entrée, de sortie, la relation fonctionelle, ou tout autre aspect que l'on veut exhiber dans les résultats d'une méta-évaluation.

La fonction META-APPLY devient un ***poseur d'equations.*** Si par exemple la définition sémantique d'une fonction de base ***FCT*** exhibe une description ***<l1 , env1>*** de son domaine d'entrée,

(META-APPLY '***FCT l2 env2***)

entre autres, pose l'équation:

<l1 , env1> = <l2 , env2>

C'est le système de raisonnement, indépendant du langage de programmation, qui se charge de résoudre cette équation.

exemple:
considérons l'exemple 1 du test LISP :

(LAST (APPEND L M))

Lors de la méta-évaluation, lorsque la méta-valeur de la sous-expression (APPEND L M) est calculée :

<(append l m) , {L = *l*,M = *m*}>

la fonction LAST y est méta-appliquée:

(META-APPLY 'LAST '***(append l m)*** {L = *l*,M = *m*})

cet appel pose tour à tour les équations:

1- ***<(append l' (cons x nil)) , env>***
   = ***<(append l m)*** , {L = *l*,M = *m*}> ①

2- ***<() , env>***

= <*(append l m)* , {L = *l*,M = *m*}>

les définitions sémantiques des fonctions META-CAR et META-LAST:

META-LAST:

1- <*(append l' (cons x nil))* , *env*> → <*x* , *env*>

2- <*()* , *env*> → <*()* , *env*>

META-CAR:

1- <*(cons x l')* , *env*> → <*x* , *env*>

2- <*()* , *env*> → <*()* , *env*>

sont courtes et non redondantes. Les raisonnements communs à leur deux défintions procédurales sont isolés dans le système de raisonnement par résolution d'équations:

le raisonnement commun à la méta-évaluation des 2 formes (LAST (APPEND L M)) et (CAR (APPEND L M)) est la résolution de l'équation :

(append a b)* = *(append c d)

Pour l'exemple présent, les équations ① se résolvent ainsi:

1- ***(append l' (cons x nil))* = *(append l m)***

donne deux solutions:

- ***m* = *(append m' (cons x nil))***
  l'* = *(append l m')

- ***m* = *()***
  l* = *(append l' (cons x nil))

2- ***()* = *(append l m)***

donne 1 solution:

- ***l* = *()***
  m* = *()

Ces trois solutions fournissent le résultat escompté.

UN SYSTEME DE REPRESENTATION DE SIGNIFICATIONS

Un système de compréhension automatique de programmes ne doit pas se contenter de décrire symboliquement la succession des états de la machine durant l'évaluation de l'appel d'un programme. Il doit fournir une ***signification*** de ce programme, objet formel à partir duquel les applications de la compréhension (diagnostics d'erreurs, extraction de propriétés utiles, annotations, vérifications, corrections) doivent être facilitées. Les représentations utilisées par le méta-évaluateur défini au chapitre 4, les listes de couples <valeur, environnement>, sont insuffisantes pour représenter des programmes LISP.

Voici un système de représentation de significations de programmes. Ces représentations sont indépendantes d'un domaine de données (par exemple le domaine des listes, en LISP, sur lequel sont définies les fonctions CAR CONS CDR NULL). Elles concernent le contrôle de l'évaluation :

- conditionelles (IF COND AND OR SELECT)
- itération (WHILE UNTIL MAPC MAPCAR DO)
- appels à l'évaluateur (EVAL)
- échappement (ESCAPE EXIT)
- appel par nom ou par valeur (DE DF)

(les exemples sont ceux du langage VLISP)

Ces représentations tiennent compte également de :

- définitions de structures de contrôle (DF et EVAL)

- arguments fonctionnels, expressions incomplètement spécifiées (macro-génération dynamique de fonctions)

- appels récursifs

- entrées-sorties (READ PRINT)

- programmation incrémentale (appels à des programmes non encore définis)

Appliqué à une expression *A* et un environnement méta-décrit *env*, et s'il termine son calcul, le méta-évaluateur retourne une liste de couples *<valeur , environnement>* tels que l'ensemble des formes [*env* → *<valeur , environnement>*] décrit la fonction dénotée par *A*. Si *A* est le corps d'un programme (λ *PFS* . *A*), alors l'ensemble des formes [*<PFS , env>* → *<valeur , environnement>*], résultat de la méta-évaluation, est la définition sémantique associée à la λ-expression.

En appliquant le méta-évaluateur sur *A* dans l'environnement le plus global construit à partir de *PFS*, on peut obtenir une définition de la λ-expression, à laquelle le méta-évaluateur peut se référer lorsqu'il a à comprendre une expression qui comporte un appel de cette

λ-expression. Si une telle définition a été obtenue, le méta-évaluateur peut évaluer simplement une application de la λ-expression sur des arguments concrets, toujours en se référant à la définition sémantique.

exemple:

à partir de la λ-expression:

(λ (L M) (LAST (APPEND L M))) ①

le méta-évaluateur obtient:

<l (append m (cons x nil)) , env> → <x , env>

<(append l (cons x nil)) () , env> → <x , env>

<() () , env> → <() , env>

Ceci est un programme au même titre que ①. Le méta-évaluateur peut simplement évaluer les appels :

(① '(a b c) '(d e)) = e

(① '(a b c) '()) = c

(① '() '(a b)) = b

(① '() '()) = ()

L'examen de cas concrets conduit à un concept de ***point de rupture*** qui permet de représenter de façon uniforme des constructions aussi diverses que:

- appels à l'évaluateur (fonction LISP EVAL)

- définition de structures de contrôle (à l'aide des fonctions LISP DF et EVAL)

- arguments fonctionels, schémas de programmes

- programmes à appel par valeur ou par nom (fonctions LISP DE et DF)

- définitions récursives

- processus interactifs, non-terminants, à base d'entrées-sorties

- programmation incrémentale. Des programmes incomplets, faisant appel à d'autres programmes non encore définis, peuvent toutefois être compréhensibles.

Exemple :

Voici la définition sémantique de (l'application de) EVAL:

```
<(EVAL a) , {Li=vi}>
        → (point-de-rupture                    ②
            <a , {Li=vi}>
            <a' , {Li=vi'}> → <a' , {Li=vi'}>)
```

La forme (point-de-rupture *exp* . *regles*) indique que *exp* est une expression à méta-évaluer. *regles* est une continuation.

Exemple :

Les deux définitions :

```
(DE FOO1 (X) X)

(DF FOO2 (L) (EVAL (CAR L)))
```

sont presque équivalentes. C'est à dire que les appels :

```
(FOO1  E)
(FOO2  E)
```

où E est une expression LISP, fournissent la même valeur, par exemple :

```
(FOO1 (CAR '(a b c))) = a
(FOO2 (CAR '(a b c))) = a
```

à l'exception d'une classe de cas, dont les appels :

```
(FOO1 (CAR L))
(FOO2 (CAR L))
```

sont un exemple.
Si L = (a b c),

```
(FOO1 (CAR L)) = a
(FOO2 (CAR L)) = (CAR L)
```

FOO1 et FOO2 reçoivent respectivement les définitions (l'atome point-de-rupture est abbrévié à PR):

```
<(FOO1 EXP) , env>
     → (PR <EXP , env>
           <x , env'>
                 → (PR <(LIER 'X 'x) , env'>
                        <anc-val , env''>
                            → (PR <(DELIER X anc-val) , env''>
                                  env''' → <x , env'''>)))
```

qui, par connaissance de LIER et DELIER, se simplifie en:

```
<(FOO1 EXP) , {L=anc-val}>
     → (PR <EXP , {L=anc-val}>
           <x , {L=v}> → <x , {L=anc-val}>)
```

et:

```
<(FOO2 EXP . Y) , env>
     → (PR <(LIER 'L '(EXP . Y)) , env>
           <anc-val , env'>
               → (PR <EXP , env'>
                     <x , env''>
                         → (PR <(DELIER L anc-val) , env''>
                               env''' → <x , env'''>)))
```

qui se simplifie en:

```
<(FOO2 EXP . Y) , {L=anc-val}>
     → (PR <EXP , {L=(EXP . Y)}>
           <x , {L=v}> → <x , {L=anc-val}>)
```

La différence entre FOO1 et FOO2 apparait à la forme (PR <EXP , *env*>. Pour FOO1, EXP est évalué dans l'environnement de départ. Pour FOO2, l'environnement contient la liaison supplémentaire L=(EXP . Y).
Ces deux définitions exhibent à la fois les points communs de FOO1 et FOO2 (l'évaluation de leur premier argument) et leurs différences (pour FOO2, cette évaluation se fait *aprés* la liaison du paramètre L).

Exemple :

CAN transforme le programme :

```
(DE FOO ()
   (F (READ))
   (FOO))
```

où F est un programme quelconque, en :

```
<(FOO) , env>
       → (PR <(READ) , env>
             <v , env'> → (PR <(F v) , env'>
                              <v' , env''>
                                  → (PR <(FOO) , env''>
                                        v'' → v'')))
```

où *v*, *v'* et *v''* sont des variables dont le domaine est l'ensemble des éléments LISP.

LE CALCUL CONCEPTUEL

Le calcul conceptuel permet de représenter sous le même statut les définitions sémantiques des unités syntaxiques de base, dont se sert le méta-évaluateur, d'un langage de programmation.

La méta-évaluation, appliquée à ces unités syntaxiques:

- manipule plusieurs contextes,

- compare des données méta-décrites (résolution d'équations).

Ces deux fonctions sont construites à priori dans l'interprète du calcul conceptuel et n'ont pas besoin d'être représentés dans la syntaxe externe.

Ecrire un programme ou un schéma de programmes en calcul conceptuel permet de le tester sur des données méta-décrites, sans qu'il soit nécéssaire de modifier le programme. Le calcul conceptuel permet la ***méta-programmation.***

exemple:
Voici un concept qui définit la fonction LAST. LAST évalue son argument, s'attend à ce que la valeur soit une liste, retourne () si elle est vide, et son dernier élément sinon.

```
<(LAST L) , env>
          → (point-de-rupture
                  <L , env>
                  <() , env'> → <() , env'>
                  <(append l (cons x nil)) , env'>
                                  → <x , env'>)
```

L'argument L de LAST peut être une expression quelconque. Le terme conceptuel <L , *env*>, premier argument du point de rupture, fait référence aux définitions conceptuelles présentes, s'il y en a. ***évaluer*** une forme (LAST expression) en calcul conceptuel, c'est donc d'abord ***évaluer*** expression, puis comparer les résultats obtenus avec chacun des termes conceptuels gauche des concepts du point de rupture: <() , ***env'***> et <***(append l (cons x nil))*** , ***env'***>. Chaque concept fournit une valeur et un environnement résultant de l'évaluation de l'expression.

La règle de composition est la principale règle de ***simplification*** du calcul conceptuel. étant donnés deux concepts [t1 → t2] et [t3 → t4], où t1, t2, t3 et t4 sont des termes conceptuels, elle transforme la composition:

[t1 → t2] o [t3 → t4]

en un ensemble de concepts de la forme:
[E → F]

Si un concept est interprété comme une procédure à invocation par ***résolution d'équation*** [note 1], alors le point de rupture peut être interprété comme:

(point-de-rupture appel-de-procédure . continuation)

La règle d'équilibration relie les concepts entre eux.

La règle du cercle vicieux :

- Du point de vue de la méta-évaluation, la règle du cercle-vicieux permet d'arrêter des calculs récursifs sur des données méta-décrites lorsqu'il est certain qu'aucune nouvelle information ne peut être obtenue.

- Du point de vue de la compréhension de programmes et de la construction de définitions à partir de programmes, la règle du cercle-vicieux permet de ne pas se contenter d'arrêter la méta-évaluation à chaque appel récursif, et de poursuivre lorsque l'appel récursif n'est pas *équivalent* à l'appel d'origine.

La règle d'induction constructrice est un rouage indispensable de la compréhension de programmes. Le système CAN obtient à partir de définitions récursives ou itératives des définitions conceptuelles dont il se sert pour effectuer des simplifications impossibles par simples pliages et dépliages des définitions récursives.

L'évaluation

Le calcul conceptuel permet d'évaluer un terme conceptuel de base dans un contexte formé d'un ensemble de concepts, de la même manière qu'il est possible, en LISP, d'évaluer une expression dans un contexte de définitions de fonctions.

Evaluer un terme de base t (sans variable) en calcul conceptuel, c'est appliquer la règle d'équilibration sur le concept:

```
[t → (PR t
         t' → t')]
```

où le concept [t' → t'] représente le fonction identité partout définie.

L'évaluation symbolique

Une évaluation intermédiaire entre cette évaluation et la méta-évaluation, l'évaluation symbolique, admet que le terme à évaluer soit un terme conceptuel quelconque. Ce terme est une donnée symbolique, ou méta-décrite. C'est la représentation symbolique d'une classe infinie de termes conceptuels de base.

La règle de composition doit résoudre des équations t=t' où t et t' sont des termes quelconques.

[note 1]: Les langages PLANNER [HEWITT 72] et CONNIVER [Mc.DERMOTT 72] ont introduit la notion de "pattern directed invocation", ou invocation par filtrage. La notion d'invocation par résolution d'équation en est une extension. Un exemple particulier d'invocation par résolution d'équation est l'invocation par unification en PROLOG [ROUSSEL 75].

Dans le cas où les termes conceptuels ne contiennent que des variables d'élément, la résolution d'équations se limite à l'unification robinsonienne [ROBINSON 65]. Les règles de composition et d'équilibration correspondent aux règles d'évaluation de clauses de HORN, en logique des prédicats, interprétées comme des procédures [VAN EMDEN 76]. La règle de résolution [ROBINSON 65], interprétée comme une règle d'invocation de procédure, correspond aux deux règles de composition et d'équilibration.

La méta-évaluation

Etant donné un concept interprété comme une forme à méta-évaluer, la règle d'équilibration peut en général s'appliquer de plusieurs manières. Elle peut s'appliquer sur le concept lui-même, et sur chacun de ses sous-concepts.
La méta-évaluation impose un choix simple pour l'application de la règle d'équilibration. Etant donnés un concept et un ensemble de choix d'application, la règle d'équilibration est appliquée sur le concept le plus extérieur. Si le concept n'est pas modifié par la règle d'équilibration, alors elle est réappliquée sur le second sous-concept le plus extérieur. S'il est modifié, le processus recommence. Ce processus termine lorsque la règle d'équilibration ne peut plus modifier le concept.

L'Analyse de programmes récursifs

Les buts d'une analyse de programmes récursifs, une fois représentés conceptuellement, sont de :

- décrire les classes de données pour lesquelles le calcul ne termine pas, ou le programme n'est pas défini.

- déceler les portions de code non utilisées, ou les chemins impossibles

- décrire de façon statique l'activité d'un programme.

- exhiber des inefficiences

- reconnaître l'équivalence sémantique profonde de deux programmes.

Exemple :

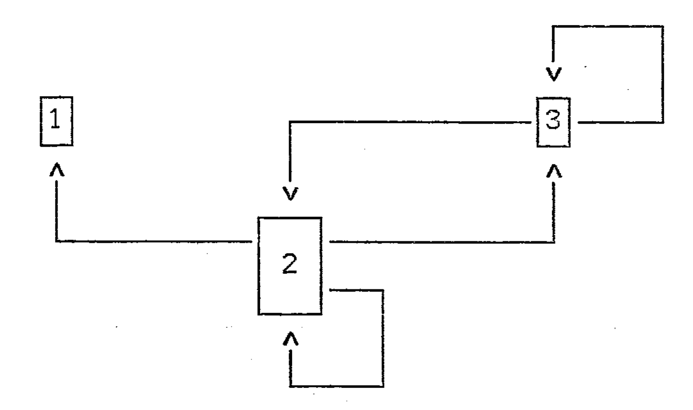
Au programme LISP qui concatène un nombre quelconque de listes :

```
(DE CONC (L)
    (COND ((NULL L) NIL)
          ((NULL (CAR L)) (CONC (CDR L)))
          (T (CONS (CAAR L)
                   (CONC (CONS (CDAR L) (CDR L)))))))
```

correspondent les trois concepts :

```
1- (CONC ()) → ()
2- (CONC (() . l)) → (PR (CONC l)
                      l' → l')                    ①
3- (CONC ((x . y) . l)) → (PR (CONC (y . l))
                               l' → (x . l'))
```

Par application de la règle d'équilibration sur chacun de ces concepts,
on obtient le graphe de dépendance :



qui exhibe 1 comme condition d'arrêt, et montre qu'aucun arc ne relie 3 à 1. Le chemin 3-1 est impossible. L'appel récursif de la troisième clause de CONC ne conduit pas directement à la condition d'arrêt de CONC.

La règle d'induction peut être appliquée sur 3, puis le résultat équilibré avec la règle 2 de CONC :

```
1- (CONC ()) → ()
2- (CONC (() . l)) → (PR (CONC l)
                      l' → l')                    ③
3- (CONC ((x1 ... xn) . l))
        → (PR (CONC l)
              l' → (x1 ... xn . l'))
```

Les résultats obtenus peuvent à nouveau être compactés en utilisant une règle d'***absorption***. La règle d'absorption réunit en un seul concept le résultat de l'induction et le concept avec lequel ce résultat est équilibré.

A la place de ③, on obtient :

```
1- (CONC ()) → ()
3- (CONC ((x1 ... xn) . l))
        → (PR (CONC l)
              l' → (x1 ... xn . l'))
```

Ce dernier exemple illustre particulièrement l'avantage de la règle d'absorption, puisque le graphe de dépendance de CONC est réduit à :



ce qui, par induction, équilibration et absorption, est compacté en :

(CONC *((x11 ... xn1) ... (x1m ... xnm))*)
→ *(x11 ... xn1 ... x1m ... xnm)*

AXIOMATISATION DES PRIMITIVES LISP

Le système CAN de compréhension automatique de programmes LISP est fondé sur les règles du calcul conceptuel: composition, équilibration, cercle-vicieux, induction constructrice.

Les règles du calcul conceptuel supposent une axiomatisation des domaines sur lesquels il est appliqué. Cette axiomatisation sert principalement la règle de composition, dont le rôle est de simplifier des expressions en concepts. Ces axiomatisations sont des systèmes de résolution d'équations.

Les axiomatisations requises par le calcul conceptuel procèdent de principes différents de ceux des axiomatisations classiques, destinées à être utilisées par des mathématiciens, ou utilisées par des systèmes de démonstration automatique. En particulier, elles ne sont pas basées sur le principe d'économie, visant à réduire au maximum le nombre d'axiomes de base. Le calcul conceptuel demande au contraire d'importants systèmes de déduction. Ces systèmes de déduction doivent minimiser les choix d'utilisation d'axiomes à chaque pas du processus de résolution d'équations.

En effet, le calcul conceptuel est censé obtenir ses résultats le plus rapidement possible, indépendamment des moyens utilisés. La résolution d'équations qu'utilise le calcul conceptuel ne doit pas s'engager dans des recherches arborescentes. Elle ne doit pas résoudre de problème. Elle constitue le pouvoir déductif de la compréhension de programmes. Ceci distingue la résolution d'équations dans le système CAN des systèmes axiomatiques qui cherchent à atteindre leurs buts avec le minimum de moyens.

La représentation de la fonction APPEND comme une fonction à deux arguments oblige à utiliser certains axiomes comme l'associativité et l'élément neutre à gauche et à droite. Ces axiomes et les déductions qu'ils permettent peuvent être ***intégrés*** dans un mode de représentation. De même peuvent y être intégrées les relations entre les fonctions CAR CDR CONS.

Nous utilisons les variables de ***séquence*** :

(APPEND L M) s'écrit (?L ?M)

où ?L et ?M ont pour domaine l'ensemble des séquences d'éléments, y compris la séquence vide.

(APPEND L NIL), par exemple, se représente (?L)

APPEND est défini ainsi :

(APPEND (?L) (?M)) = (?L ?M)

Cette représentation automatise l'utilisation de l'associativité-élément neutre de APPEND, et par l'utilisation de variables d'élément, distinguées par un "!", elle permet de définir CAR CDR CONS indépendamment les unes des autres:

```
(CAR  (!A ?B)) = !A
(CDR  (!A ?B)) = (?B)
(CONS !A (?B)) = (!A ?B)
```

Cette représentation uniforme des fonctions CAR CDR CONS APPEND à partir de filtres permet de réduire les systèmes de simplification et de résolution d'équations à la comparaison de filtres [GOOSSENS 78c].

L'utilisation de variables d'élément typées permet de définir conceptuellement les primitives d'un langage de programmation qui servent de connecteurs logiques. En LISP, par exemple, on peut définir:

```
(AND !X1 ... !Xn) = si (META-EVAL !X1)=NIL alors NIL
                    sinon
                       (AND !X2 ... !Xn)
(AND) = T

(OR !X1 ... !Xn) = si (META-EVAL !X1)=NIL
                      alors (OR !X2 ... !Xn)
                   sinon
                      (META-EVAL !X1)=!Y
                      !Y
(OR) = NIL

(IMPLIES !X !Y) = si (META-EVAL !X)=NIL alors T
                  sinon
                     (META-EVAL !X)=!Z
                     (META-EVAL !Y)

(NOT !X) = si (META-EVAL !X)=NIL alors T
           sinon NIL
```

(les variables soulignées ne peuvent être associées à la valeur LISP NIL. Du point de vue de la méta-évaluation, les définitions ci-dessus n'exhibent que le contrôle que les fonctions ont sur l'évaluation de leurs arguments. Les modifications qu'elles entrainent sur l'environnement courant ne sont pas considérées).

Les variables d'élément typées permettent également de définir les

prédicats de base (ex: ATOM, NUMBP, LISTP, STRING en LISP).

exemple:

```
(ATOM  !atom) = T
(ATOM  !~atom) = NIL
```

où !atom et !~atom sont respectivement une variable dont le domaine est l'ensemble des atomes LISP et une variable dont le domaine est l'ensemble des éléments non atomiques.

Ces accomodations permettent d'étendre les résultats théoriques de l'unification de chaines aux expressions formées à partir des primitives CAR CDR CONS NOT ATOM NUMBP STRING APPEND AND OR IMPLIES EQUAL.

La définition de APPEND:

```
(APPEND  (?A) (?B)) = (?A ?B)
```

améliore la définition classique:

```
(APPEND NIL L) = L
(APPEND (CONS X Y) L) = (CONS X (APPEND Y L))
```

qui ne permet pas de simplifier les formes :

```
(APPEND L NIL)
(EQUAL (APPEND L M) L)
(EQUAL (APPEND L M) NIL)
(EQUAL (APPEND L M) (APPEND L N))
(EQUAL (APPEND L L) (APPEND M M))
```

de même, la définition:

```
(REVERSE NIL) = NIL
(REVERSE (CONS X Y)) = (APPEND (REVERSE Y) (CONS X NIL))
```

ne permet pas de simplifier les formes :

```
(REVERSE (APPEND X (CONS Y NIL)))
(REVERSE (APPEND L M))
(REVERSE (REVERSE X))
(EQUAL (REVERSE X) X)
```

elle est améliorée par cette définition, utilisée par le système CAN:

```
(REVERSE (!X1 ... !Xn)) = (!Xn ... !X1)
```

qui s'écrit:

$$(\mathrm{REVERSE}\ (<\mathrm{L}_{i}^{n}\ !X_{i}>)) = (<\mathrm{L}_{i}^{n}\ !X_{n-i+1}>)$$

Ce genre de définition demande à accepter un nouveau type de variable pour construire des filtres, les notations indicées.

Les notations indicées

Pour définir conceptuellement des fonctions courantes de manipulation de listes comme les fonctions LISP REVERSE, MEMBER, ASSOC, NTH, LENGTH, il faut utiliser un nouveau type de variable dont la syntaxe est:

<L^{p+}_{indice} FILTRES>

Pour admettre ce nouveau type de variable, il faut également admettre des filtres numériques (p+ doit être un filtre numérique) et s'attendre à ce que les variables d'élément et de séquence soient indicées. Les filtres numériques sont exposés au paragraphe 8.2.6.

Cette notation indicée utilisée par le système de résolution d'équations défini au paragraphe 8.2.5.3, se définit algébriquement comme le type abstrait [GUTTAG 77, BERT 79] suivant:

type VSC (variable de séquence contrainte)

spécification syntaxique:

GEN : indice*filtre-numérique*filtre → VSC
PREMIER : VSC → filtre
COUPE-DROITE : VSC*filtre-numérique → VSC
COUPE-GAUCHE : VSC*filtre-numérique → VSC
ETENDRE : VSC*entier-positif → VSC
DECALAGE-CIRCULAIRE : VSC*entier-positif → VSC

sémantique:

(PREMIER (GEN i n p)) = $p\Big|_{i}^{0}$

(COUPE-DROITE (GEN i n p) m) = (GEN i n-m $p\Big|_{i}^{i+m}$)

(COUPE-GAUCHE (GEN i n p) m) = (GEN i m p)

(ETENDRE (GEN i n p) a)

=(GEN i n/a $p\Big|_{i}^{a*i}$ $p\Big|_{i}^{a*(i-1)}$ ... $p\Big|_{i}^{a*(i-(a-1))}$)

(DECALAGE-CIRCULAIRE (GEN i n p) a)

= si a<|p|

((npremiers a p)$\Big|_{i}^{0}$

(GEN i n-1 (saufpremiers a p)

(npremiers a p)$\Big|_{i}^{i+1}$)

(saufpremiers a p)$\Big|_{i}^{n-1}$))

NOMBRES

Les représentations conceptuelles de listes abstraites à l'aide de notations indicées exhibent des *longueurs abstraites* sous la forme de filtres numériques, c'est-à-dire à interpréter comme des classes d'entiers positifs.

Les manipulations d'entiers en LISP conduisent à des problèmes de compréhension qui reviennent le plus souvent à décider de la valeur d'une forme:

(relation expl ... expn)

où relation est une des relations de base en LISP: <, >, <=, >=, =. et où les expi sont des expressions numériques.

Du point de vue de la compréhension de programmes de traitement de listes, il faut particulièrement s'attacher au cas où les expi sont des expressions sur les entiers positifs, formés à partir de l'addition et la soustraction.
En choisissant les définitions conceptuelles:

```
(PLUS X1 ... Xn) = X1+ ... +Xn
(DIFFER X+Y Y) = X
(DIFFER X  X+Y+1) = 0
(<=  X  X+Y) = T
(<=  X+Y+1  X) = NIL
(>=  X  X+Y+1) = NIL
(>=  X+Y  X) = T
```

```
(<  X   X+Y+1) = T
(<  X+Y   X) = NIL
(>  X   X+Y) = NIL
(>  X+Y+1   X) = T
```

Le problème auquel la résolution d'équations doit faire face se réduit à la résolution d'équations du type:

$$a_1X_1 + a_2X_2 + \ldots + a_nX_n + a_{n+1} = b_1Y_1 + \ldots + b_mY_m$$

où les X_i et les Y_i sont des variables dont le domaine est celui des entiers positifs et où les a_i et les b_i sont des coefficients numériques constants, entiers et positifs.

Ce problème est un problème de résolution d'équations diophantiennes [STICKEL 75, HUET 78]. L'algorithme d'unification qu'utilise actuellement le système CAN n'est pas minimal. Il consiste en l'unifieur de séquences abstraites (règle 13, *SUIVANT*, *SUIVANT CONDITIONEL*) auquel s'ajoute une règle de simplification qui élimine les variables et constantes communes aux deux termes à unifier. Cet algorithme d'unification tient donc compte des propriétés de commutativité, associativité, élément neutre de +.

LE SYSTEME CAN

Le système CAN est un système de compréhension de programmes LISP, implémenté en VLISP [GREUSSAY 77, CHAILLOUX 80] sur un PDP KL-10. Il est basé sur les processus de méta-évaluation et de construction de significations exposés dans cette étude.
CAN fournit une description fonctionelle (sans effet de bord) des programmes qui lui sont soumis, et en simplifie la structure de contrôle. Cette description prend la forme d'un ensemble de concepts (éléments du calcul conceptuel défini au chapitre 7). Un programme est ainsi divisé, par l'analyse de cas opérée par méta-évaluation, en modules interdépendants. CAN construit et analyse le graphe de dépendance de cet ensemble de modules.
Cette description modulaire est ensuite compactée par induction, autant que le permettent les notations spéciales utilisées par CAN, et son pouvoir de résolution d'équations (chapitre 8).

En analysant leur graphe de dépendance, CAN est capable d'extraire des programmes qui lui sont soumis :

- des portions de code non utilisées

- des chemins impossibles

- des classes de données pour lesquelles le programme ne termine pas son calcul

- des redondances

Exemples :

Le programme MEMLONG suivant vérifie si deux listes ont la même longueur :

```
(DE MEMLONG (L M)
    (COND ((NULL L) (NULL M))
          ((NULL M) NIL)
          (T (MEMLONG (CDR L) (CDR M)))))
```

réponse de CAN :

```
=   (MEMLONG !L  !M ) →
=
=
=       Point-de-rupture [ !L  !M ]
=
=           (NIL !5 ) → NIL
=           ( !3 NIL) → NIL
=           (( (L<i ?N3+>  !9i )) ( (L<i ?N3+>  !14i ))) → T
=           (( (L<i ?N2+>  !9i ))
=            ( (L<i ?N2+>  !14i ) ?47 )) → NIL
=           (( (L<i ?N2+>  !9i ) ?54 )
=            ( (L<i ?N2+>  !14i ))) → NIL
```

Voici une définition incomplète de la fonction MEMBER :

```
?  (DE MEM (X L)
?     (IF (EQUAL X (CAR L))
?         L
?         (MEM X (CDR L))))
=  MEM
?
?  (CAN 'MEM)

=   (MEM !X  !L ) →
=
=
=       Point-de-rupture [ !X  !L ]
=
=           ( NIL
=           ( (L<i ?N2+>   !161i  / (NIL ≠ !161i ))) ) → NIL
=
=           (  !162
=           ( (L<i ?N3+>   !161i  / ( !162  ≠ !161i ))) )
=              → ( !162  NIL) : CALCUL INFINI
=
=           (  !166
=           ( (L<i ?N4+>  !161i /( !166 ≠ !161i )) !166 ?168 ) )
=              → ( !166  ?168 )
```

La classe de données pour lesquelles le calcul ne termine pas peut être décrite par : "les cas où X n'est pas un élément de la liste L" mais cette description exclut le cas particulier, que CAN décèle, où X a la valeur NIL. Dans ce cas, le calcul termine aprés que MEMBER ait parcouru toute la liste.

POUVOIR REPRESENTATIF

CAN utilise les résultats de la méta-évaluation d'un programme pour en construire une définition.
Pour mettre la méta-évaluation au service de la compréhension de programmes, nous avons dû généraliser les notions déjà connues d'évaluation symbolique, ou exécution symbolique. Cette généralisation a ainsi permis de tenir compte des fonctions LISP EVAL, SET et DF permettant un style de programmation particulièrement riche, puisque l'utilisateur peut définir et utiliser ses propres structures de contrôle, et écrire des programmes qui construisent et utilisent d'autres programmes.

Nous avons, par le calcul conceptuel, défini une base de représentation équivalente aux clauses de HORN [KOWALSKI 73], fondement du langage PROLOG [ROUSSEL 75], mais ouverte vers les extensions nécessaires à la compréhension de programmes :

- notations spécialisées évitant l'emploi de règles de simplification fréquemment utilisées

- définitions récursives compactables par induction et utilisables comme règles de simplification

Ces extensions nous ont permis d'appliquer le calcul conceptuel, dans le cadre de CAN, à la compréhension de programmes LISP. CAN est ainsi capable d'associer à un programme LISP une définition du calcul conceptuel, puis, si elle comporte des appels récursifs, de la compacter, c'est à dire de la débarasser de ses appels récursifs pour la rendre utilisable en tant que règle de simplification.

Exemple :

A partir de l'axiomatisation des fonctions CAR CDR CONS APPEND NULL :

```
(CAR (CONS X Y)) = X
(CDR (CONS X Y)) = Y

(NULL ()) = T
(NULL (CONS X Y)) = NIL

(APPEND () L) = ()
(APPEND (CONS X Y) L) = (CONS X (APPEND Y L))
```

et des règles d'inférence du calcul conceptuel, CAN peut, directement, prouver les théorèmes :

```
(APPEND X (APPEND X X)) = (APPEND (APPEND X X) X)
[(APPEND X X) = (APPEND Y Y)] => X = Y
```

mais aussi simplifier les expressions :

```
(APPEND X (APPEND Y Z)) = (APPEND (APPEND X W) Z)
                 en Y = W
(APPEND L M) = (APPEND L N)
```

en M = N
(APPEND L L) = (APPEND M M)
en L = M

POUVOIR DEDUCTIF

Le calcul conceptuel sans notations spécialisées (variables de séquence, notations indicées) offre un pouvoir représentatif équivalent à LISP. Cependant, il faut introduire des notations spécialisées pour développer le pouvoir déductif.
Le pouvoir déductif de CAN prend la forme de systèmes de résolution d'équations qui tiennent compte de la présente des notations spécialisées. C'est l'extension par ajout de notations spécialisées et de systèmes de résolution d'équations qui différencie CAN des systèmes de vérification et de compréhension de programmes fondés sur la logique des prédicats.
Cette voie est à rapprocher des tentatives de remplacement, dans un démonstrateur de théorèmes, d'axiomes fréquement utilisés (comme les propriétés d'associativité et de commutativité) par des procédures d'unification spécialisées [PLOTKIN 72, STICKEL 75, SIEKMANN 75]. Nous avons contribué à ces tentatives par l'ajout de notations spécialisées qui tiennent compte des propriétés particulières des fonctions LISP **REVERSE**, **MEMBER**, **ASSOC**, **NTH**, **LENGTH**, **LAST**. CAN est également capable, à partir d'une définition LISP, d'en dériver une description à base de ces notations spécialisées et donc d'utiliser son pouvoir déductif pour en comprendre les utilisations.

CAN peut être vu dans son état actuel, comme une première tentative d'adjoindre à tout système à trés longue durée de vie, une composante d'auto-compréhension nécessaire pour doter ce système d'une large capacité d'auto-maintenance.
Ceci peut donner à penser que l'action ponctuelle et dominante du programmeur actuel sur un système à construire ou à améliorer, fera place à trés long terme, à une interaction programmeur-système, où l'initiative du système pour les décisions concernant sa conception sera de plus en plus importante.

REFERENCES

[AUBIN 77]
AUBIN R. 1977
Strategies for mechanizing structural induction
Proc. 5th. IJCAI. MIT. Cambrige Mass. Aug 1977. pp 363-369.

[AUBIN 77b]
AUBIN R. 1977
Mechanizing structural induction
PH.D. Thesis Dept. of AI Univ. of Edinburgh, Scotland. 1977

[BALZER 77]
BALZER R. GOLDMAN N. WILE D. 1977
Meta-evaluation as a tool for program understanding
5th IJCAI, MIT Cambridge, August 1977

[BERT 79]
BERT D. 1979
Spécification algébrique des types abstraits et certification de programmes
AFCET Bulletin GROPLAN n.8 1979.

[BERZINS 78]
BERZINS V. 1978
Abstract model spécifications for data abstractions
PH.D.Thesis Draft. MIT Lab for comp. science.

[BORRAS 80]
BORRAS P. 1980
AIDE: un systeme d'évaluation symbolique du langage LISP
Rapport de stage. Univ. PARIS 6. Juin 80.

[BOYER 75a]
BOYER R. S. MOORE J. S. 1975
Proving theorems about LISP functions
JACM Vol. 22 n.1 pp. 129-144. 1975.

[BOYER 75b]
BOYER R.S. ELSPAS B. LEVITT K.N. 1975
SELECT--A formal system for testing and debugging programs by symbolic execution
Int. Conf. on Reliable Software, Los Angeles, April 1975, pp. 234-245.

REFERENCES

[BOYER 77]
BOYER R. S. MOORE J. S. 1977
A computer proof of the correctness of a simple optimizing compiler of expressions
Tech. rep. N00014-75-c-0816-SRI-4079. SRI International. Menlo Park. California. Jan 77.

[BURSTALL 69]
BURSTALL R.M. 1969
Proving properties of programs by structural induction
Computer Journal. Vol 12. n.1. Fev 69. pp 41-48.

[BURSTALL 72]
BURSTALL R.M. 1972
Some techniques for proving correctness of programs which alter data structures
Machine Intelligence 7. Michie ed. New York.

[BURSTALL 77]
BURSTALL R.M. DARLINGTON J. 1977
Some transformations for developping recursive programs
JACM Vol 24. n.1. Jan 77.

[CARTWRIGHT 76]
CARTWRIGHT R. 1976
A practical formal semantic definition and verification system for typed LISP
Memo-aim 296, Stanford Univ. Dept. of Comp. Science. Dec 76.

[CHAILLOUX 78a]
CHAILLOUX J. 1978
VLISP-8 un systeme LISP pour micro-processeur à mots de 8 bits
RT-21-78, Dept d'informatique. Univ. PARIS 8. Juillet 78.

[CHAILLOUX 78b]
CHAILLOUX J. 1978
a VLISP interpreter on the VCMC1 machine
LISP bulletin n.2, july 78.

[CHAILLOUX 78c]
CHAILLOUX J. 1978
VLISP-10 manuel de référence
RT-16-78, Dept d'informatique. Univ. PARIS 8. aout 78.

[CHAILLOUX 80]
CHAILLOUX J. 1980
Le modele VLISP: description, implementation, evaluation.
Thèse de 3ième cycle, Univ P. et M. CURIE.

REFERENCES

[CHARNIAK 72]
CHARNIAK E. 1972
Towards a model of children's story comprehension
AI-TR-266 MIT, AI-LAB, Dec. 1972.

[CHARNIAK 75]
CHARNIAK E. 1975
Organisation and inference in a frame-like system of common-sense knowledge
Working Paper 14. Institute for Semantic and Cognitive Studies. Castagnola, Switzerland 1975.

[CHEATHAM 79]
CHEATHAM T.E., HOLLOWAY G.H., TOWNLEY J.A. 1979
Symbolic evaluation and the analysis of programs
IEEE transactions on software engineering, vol SE-5, n.4, July 1979, pp 402-417.

[CHURCH 41]
CHURCH A. 1941
The calculi of λ-conversion
Annals of Math. studies n.6. Princeton Univ. Press. Princeton 1941.

[COOK 75]
COOK A. OPPEN D.C. 1975
An assertion language for data structures
2nd. ACM Symposium on principles of programming languages. Palo Alto California Jan. 75

[DURIEUX 78]
DURIEUX J.L. SALLE P. 1978
Application de la notion d'échappement à la description des instructions de saut
AFCET, Bulletin GROPLAN n.5 1978.

[FICKAS 79]
FICKAS S. BROOKS R. 1979
Recognition in a program understanding system
6th. IJCAI. Tokyo. Aug. 79. pp 266-268.

[FLOYD 67]
FLOYD R.W. 1967
Assigning meanings to programs
Math. Aspects of Comp. Science Proc. Symp. in Applied Math. 19, Providence (RI), Amer. Math. Soc. 1967, Schwartz ed.

[GERHART 76]
GERHART S. 1976
Proof theory of partial correctness verification systems
SIAM Journal Comput. Vol 5. n.3. Sept 76.

REFERENCES

[GOOSSENS 77]
GOOSSENS D. 1977
CAN: un systeme de méta-interprétation du langage LISP
Dept. d'Informatique, Univ. Paris 8-Vincennes, 1977.

[GOOSSENS 78a]
GOOSSENS D. 1978
A system for visual-like understanding of LISP programs
A.I.S.B. Conf. Hamburg, July 1978.

[GOOSSENS 78b]
GOOSSENS D. 1978
Compréhension visuelle de programmes controlée par méta-filtrage
Groplan: Bulletin de l'AFCET, Groupe Programmation et langages, 1978

[GOOSSENS 78c]
GOOSSENS D. 1978
L'unification au service de la compréhension
RT-14. Dept. d'informatique, Université de Vincennes.

[GOOSSENS 79]
GOOSSENS D. 1979
Meta-interpretation of recursive list-processing programs
6th. IJCAI. Tokyo. Aug 79.

[GREEN 75]
GREEN C. BARSTOW D. 1975
A hypothetical dialogue exhibiting a knowledge base for a program understanding system
Stanford A.I. Lab. Memo-Aim 258, Cpt Science Dept. Report n. Stancs-75-476.

[GREUSSAY 76]
GREUSSAY P. 1976
VLISP: structure et extension d'un systeme LISP pour mini-ordinateur
RT-16-76, UER Informatique et linguistique, Univ PARIS 8 Vincennes, Janvier 76.

[GREUSSAY 77]
GREUSSAY P. 1977
Contribution a la définition interprétative et à l'implémentation des λ-langages
Thèse, Université PARIS 7. Nov. 77.

[GREUSSAY 78]
GREUSSAY P. 1978
Le systeme VLISP-16
Ecole polytechnique. Décembre 78.

REFERENCES

[GREUSSAY 79]
GREUSSAY P. 1979
VLISP-11 manuel de référence
Université PARIS 8 Vincennes. 1979.

[GREUSSAY 80]
GREUSSAY P. 1980
Program understanding by reduction sets
7th. AISB. Amsterdam. 1980.

[GUTTAG 77]
GUTTAG J. 1977
Abstract data types and the development of data structures
Comm. of ACM. Vol. 20 n.6 Juin 1977. pp 396-404.

[HEWITT 69]
HEWITT C. 1969
PLANNER: A language for manipulating models and proving theorems in a robot
1st. IJCAI. Washington DC.

[HEWITT 72]
HEWITT C. 1972
Description and theoritical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot
MIT Revised Ph.D. Dissertation. AI-TR-258. April 72.

[HEWITT 75]
HEWITT C.E. SMITH B. 1975
Towards a programming apprentice
IEEE Trans. on soft. engineering, Vol. SE-1. pp. 26-45.

[HMELEVSKII 66]
HMELEVSKII J.I. 1966
Word équations without coefficients
Soviet Math Dokl. Vol 7 n.6 1966.

[HMELEVSKII 67]
HMELEVSKII J.I. 1967
Solution of word équations in three unknowns
Soviet Math Dokl. Vol 8 n.6 1967.

[HOARE 69]
HOARE C.A.R 1969
An axiomatic basis for computer programming
Comm. ACM, Vol 12, n.10, Oct. 1969. pp 576-583.

[HUET 75]
HUET G. 1975
A unification algorithm for typed λ-calculus
Theoritical Comp. Science 1, 1975.

REFERENCES

[HUET 77]
HUET G. 1977
Résolution d'équations dans les langages d'ordre 1, 2, ... omega
Thèse. Université PARIS 7. Sept. 77.

[HUET 78]
HUET G. 1978
An algorithm to generate the basis of solutions to homogeneous linear diophantine equations
Rapport de recherche n.274. Jan. 1978.

[IGARASHI 75]
IGARASHI LONDON LUCKHAM 1975
Automatic program verification I: A logical basis and its implementation
Acta Informatica, Vol. 4, pp. 145-182.

[KARR 76]
KARR M. 1976
Summation in finite terms
Mass. Computer Associates. Wakefield MA. Tech. Rep. Feb. 76

[KATZ 73]
KATZ S. MANNA Z. 1973
A heuristic approach to program verification
Proc. 3rd. Int. Conf. on AI. Palo-alto. California. 73. pp 500-512.

[KING 75]
KING J. 1975
A new approach to program testing
Int. Conf. on reliable software, April 1975, pp. 228-233.

[KING 76]
KING J. 1976
Symbolic execution and program testing
Comm. of ACM. Vol 19. n.7 July 76. pp 385-394

[KOWALSKI 73]
KOWALSKI R. 1973
Predicate logic as a programming language
Memo n.70, Dept. of Comp. Logic School of AI. Univ. of Edinburgh. Nov. 73.

[KUHNER 77]
KUHNER S. MATHIS C. RAULEFS P. SIEKMANN J.
Unification of idempotent functions
5th. IJCAI. MIT. Cambrige. Aug 77. p 528.

REFERENCES

[LENTIN 72]
LENTIN A. 1972
Equations dans les monoides libres
Gauthier Villars, Paris 1972.

[LIEVESEY 76]
LIEVESEY M. SIEKMANN J. 1976
Unification of bags and sets
Internal Report 3/76, Institut fur Informatic I, Univ. Karlsruhe.

[LISBUL 80]
LISP Bulletin n.2
P. Greussay et J. Laubsch eds. Univ. Paris 8, Dept. d'informatique. 1980.

[LISKOV 75]
LISKOV B.H. ZILLES S.
Specifications techniques for data abstractions
Proceedings of ACM Int. Conf. on Reliable Software. Los Angeles 1975.

[LISKOV 77]
LISKOV B.H. BERZINS V.
An appraisal of program specifications
MIT Lab. for Comp. Science. Memo 141-1 1977.

[LONDON 74]
LONDON R.L. MUSSER D.R. 1974
Application of a symbolic mathematical system to program verification
Proc. of ACM. 1974. pp 265-273.

[MACSYMA 75]
MACSYMA reference manual
Mathlab Group, Proj. MAC, MIT. Cambrige Mass.

[MAKANIN 77]
MAKANIN G.S. 1977
The problem of solvability of equations in a free semi-group
Soviet Math. Dokl. Vol 18 n.2 1977.

[MANNA 73]
MANNA Z. 1973
Inductive methods for proving properties of programs
Comm. of ACM. Vol 16. n.8. Aug 73. pp 491-502

[Mc. DERMOTT 72]
Mc. DERMOTT D.V. SUSSMANN G.J. 1972
The CONNIVER reference manual
AI-memo 259a. MIT AI-Lab. May 1972.

REFERENCES

[MEYER 72]
MEYER G.S. 1972
Infants in children stories. Towards a model of natural language comprehension
MIT AI-Lab. Aug. 72. Memo 265.

[MINSKY 75]
MINSKY M.
A framework for representing knowledge
In Ph. Winston ed. The psychology of computer vision. New York, Mc. Graw Hill 1975, pp 211-277.

[MOORE 73]
MOORE J. NEWELL A. 1973
How can MERLIN understand ?
Dept. of Comp. Science, Carnegie Mellon Univ. Pittsburgh Pennsylvannia, Nov. 73.

[OPPEN 78a]
OPPEN D.C. 1978
Reasonning about recursively defined data structures
5th Annual ACM symp. on principles of programming languages. jan 78. pp 151-157.

[OPPEN 78b]
OPPEN D.C. NELSON G. 1978
Simplification by cooperating decision procedures
Memo-aim-311 report n. STAN-CS-78-652. SAIL. April 78.

[OPPEN]
OPPEN D.C.
Tools for program analysis. Oppen.sli, Stanford Univ.

[PARK 69]
PARK D. 1969
Fixpoint induction and proof of program properties
Machine intelligence 5. Meltzer and Michie eds. Edinburgh Univ. Press. 1969. pp 59-78.

[PLOTKIN 72]
PLOTKIN G.D. 1972
Building-in equational theories
Mach. Int. 7, Meltzer Michie eds., 1972.

[RAPHAEL 71]
RAPHAEL B. 1971
The frame problem in problem-solving systems
Artificial Intelligence and Heuristic Programming. Findler and Meltzer eds. Edinburgh Univ. Press. 1971.

REFERENCES

[RAULEFS 78]
RAULEFS P. SIEKMANN J. SZABO P. UNVERICHT E. 1978
A short survey on the state of the art in matching and unification problems
Institut fur Informatik I. Univ. Karlsruhe.

[REDDY 75]
REDDY R. 1975 ed.
Speech recognition: invited papers presented at the IEEE Symposium
Academis Press, New York.

[REYNOLDS 72]
REYNOLDS J.C. 1972
Definitional interpreters for higher order programming languages
Proc. of 25th ACM National Conf. Boston, 1972.

[RICH 75]
RICH C. SCHROBE H.E. 1975
Understanding LISP programs: Towards a programming apprentice
Master's Thesis, EECS M.I.T.

[RICH 76]
RICH C. SCHROBE H.E. 1976
Initial report on a LISP programmer's apprentice
MIT AI-TR-354 Dec. 1976.

[RICH 79]
RICH C. SCHROBE H.E. WATERS R.C. 1979
Overwiew of the programmer's apprentice
6th. IJCAI. Tokyo. Aug 79. pp 827-828.

[ROBINET 78]
ROBINET B. 1978
Petit précis de λ-calcul
Implémentation et interprètation de LISP. Ecole IRIA. Toulouse. pp 15-24. Mars 1978.

[ROBINSON 65]
ROBINSON J. 1965
A machine-oriented logic based on the resolution principle
Journal of Assoc. for Comp. Machinery. Vol 12. n.1. Jan 1965. pp 23-41.

[ROUSSEL 75]
ROUSSEL P. 1975
PROLOG manuel de référence et d'utilisation
Groupe d'Intelligence Artificielle. UER Marseille-Luminy. Sept 75.

REFERENCES

[RUTH 74]
RUTH G.R. 1974
Analysis of algorithm implementation
MIT. MAC-TR-130.

[SCHANK 75a]
SCHANK R.C. GOLDMAN N. RIEGER C.J. RIESBECK C. 1975
MARGIE: Memory, analysis, response generation, and inference on english
Stanford Univ. 1973

[SCHANK 75b]
SCHANK R.C. 1975
Conceptual information processing
Noth Holland, Amsterdam.

[SCHANK 77]
SCHANK R.C. ABELSON R. 1977
Scripts, plans, goals and understanding
Lawrence Erlbaum Press, Hillsdale NJ.

[SCHUTZENBERGER 66]
SCHUTZENBERGER M.P. 1966
Quelques problemes combinatoires de la théorie des automates
Chap. II.1, Univ. Paris 6, Institut de Programmation, 1966.

[SELFRIGE 59]
SELFRIGE O. 1959
Pandemonium: a paradigm for learning
Symp. on the mechanization of thought processes. London HM Stationery Office.

[SETHI 77]
SETHI R. 1977
Semantics of computer programs: overview of language definition methods
Bell Laboratories, Murray Hill, New Jersey 07974.

[SHROBE 79]
SHROBE H.E. 1979
Dependency directed reasonning in the analysis of programs which modify complex data structures
6th. IJCAI. Tokyo. Aug 79. pp 829-835.

[SIEKMANN 75]
SIEKMANN J. 1975
String unification
Essex university, Memo CSM-7.

REFERENCES

[SIEKMANN 75b]
SIEKMANN J. LIVESEY M. 1975
Termination and decidability results for string unification
Essex university, CSM-12.

[SMITH 75]
SMITH B.C. HEWITT C. 1975
A plasma primer
MIT, AI-Lab, Draft. Sept 1975.

[STEELE 76]
STEELE G.L. SUSSMAN G.J.
Lambda, the ultimate imperative
AI-memo 353 MIT AI-Lab. March 76.

[STICKEL 75]
STICKEL 1975
A complete unification algorithm for associative-commutative functions
4th Ijcai, Tbilisi Georgia USSR. sept. 1975.

[TENNENT 76]
TENNENT R.D. 1976
The denotational semantics of programming languages
Comm. of ACM, Aug 76, Vol. 19, n.8, pp 528-535.

[VAN EMDEN 76]
VAN EMDEN et KOWALSKI R.A. 1976
The semantics of predicate logic as a programming language
JACM. Vol 23. n.4. Oct 1976. pp 733-742.

[WALDINGER 74]
WALDINGER R. LEVITT K.N. 1974
Reasonning about programs
Art. Int. Vol 5 n.3, North-holland Amsterdam pp 235-316.

[WALTZ 79]
WALTZ D.L. BOGESS L.
Visual analog representations for natural language understanding
6th. IJCAI, Tokyo. pp 926-934 Aug.1979.

[WARREN 77]
WARREN H.D. 1977
Implementing PROLOG
Vols. 1 et 2, DAI Research Report n.40, Univ. of Edinburgh, Dept of AI, 1977.

[WATERS 78]
WATERS R.C. 1978
Automatic analysis of the logical structure of programs
Ph.D. Thesis. MIT. AI-Lab. 1978.

REFERENCES

[WATERS 79]
WATERS R.C. 1979
A method for automatically analysing programs
6th. IJCAI. Tokyo. Aug. 1979. pp 935-941.

[WEGBREIT 73]
WEGBREIT B. 1973
Heuristic methods for mechanically deriving inductive assertions
Proc. 3rd. IJCAI, Aug 1973, pp 524-536.

[WEGBREIT 76]
WEGBREIT B. SPITZEN J.M. 1976
Proving properties of complex data structures
Journal of the Assoc. for Computing Machinery, Vol 23 n.2, April 76 pp 389-396.

[WERTZ 78a]
WERTZ H. 1978
Correction automatique de programmes LISP
Journées SESORI sur la programmation. St. REMY de Provence. Mai 78.

[WERTZ 78b]
WERTZ H. 1978 *Un systeme de compréhension, de programmes incorrects*
Proc. 3ème colloque sur la programmation, Paris B. Robinet éd. pp 31-49.

[WERTZ 78c]
WERTZ H. 1978
Un systeme de compréhension, d'amélioration, et de correction de programmes incorrects
Thèse de 3ème cycle, Univ. P. et M. CURIE.

[WINOGRAD 73]
WINOGRAD T. 1973
Breaking the complexity barrier (again)
Proc. of the ACM. SIGIR-SIGPLAN interface meeting. Nov 73.

[YONEZAWA 75]
YONEZAWA Akinori 1975
meta-evaluation of actors with side-effects
Working paper 101, MIT AI-Lab, june 1975.

[YONEZAWA 76]
YONEZAWA A. HEWITT C. 1976 *Symbolic evaluation using conceptual representations for programs with side-effects*
M.I.T., A.I. Lab., AI-Memo 399. Dec. 76.

REFERENCES

[YONEZAWA 77]
YONEZAWA A. 1977
Specification and verification techniques for parallel programs based on message passing semantics
PH.D.Thesis, MIT Lab for Comp. Science, MIT-LCS-TR-191. Dec. 1977.

MULTI-FENETRAGE DYNAMIQUE

Patrick GREUSSAY

Département d'Informatique
Université Paris VIII - Vincennes
75571 PARIS Cedex 12
&
C.N.R.S. LA 248 L.I.T.P.
2 place Jussieu
75005 Paris

RT-5-80

Mai 1980

Resumé :

Nous présentons les algorithmes de base permettant la réalisation de systèmes visuels d'entrées-sorties en multi-fenêtres. Ces systèmes sont interactifs et doivent permettrent à leur utilisateur de reconfigurer à tout instant son dispositif multi-fenêtres de façon continue. Sous un tel système, l'écran de visualisation doit apparaître comme un plan de travail virtuel composé de documents multiples créables et déplaçables à volonté. Les algorithmes présentés visent à rendre, lors des déplacements de fenêtres, les transitions écran-écran les plus continues possible visuellement. Ces algorithmes ont permis, sur PDP11, la mise en place d'un tel système: WIN.

MOTS-CLES:

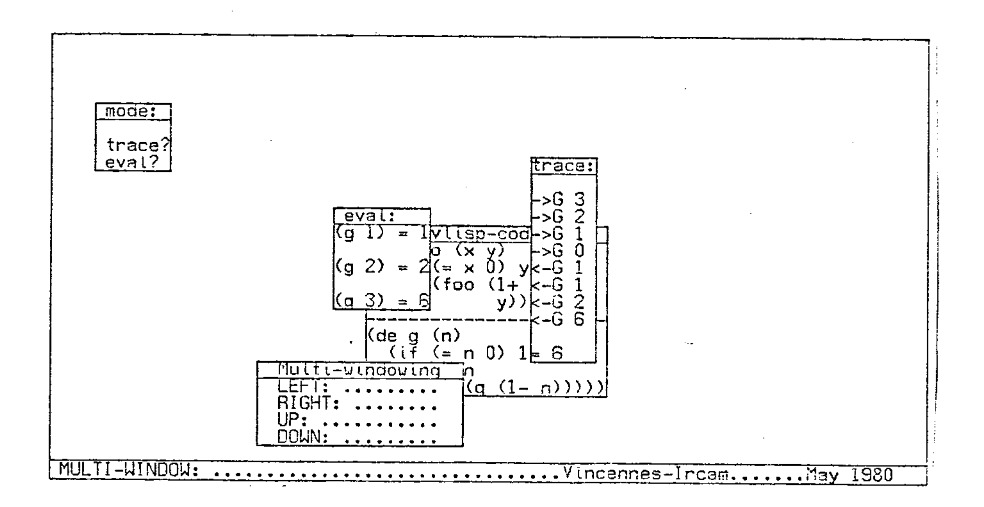
graphique, fenêtres, queue de priorité, bureautique, menus, éditeurs temps-réel, display-vecteurs, VLISP, représentations multiples.

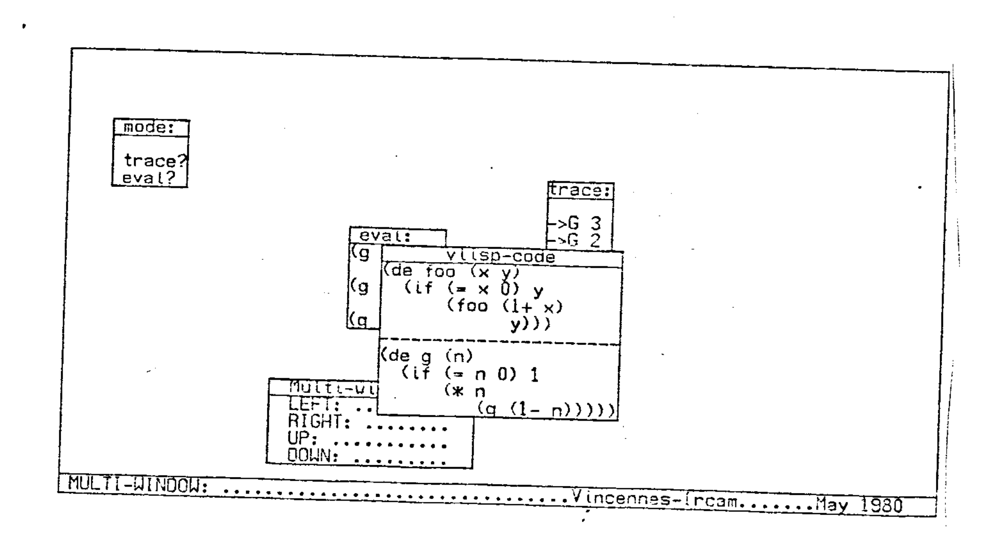
L'adjonction à la vision d'un bord arbitraire souligne le contraste entre la limitation de la perception humaine et l'infinité de Dieu.

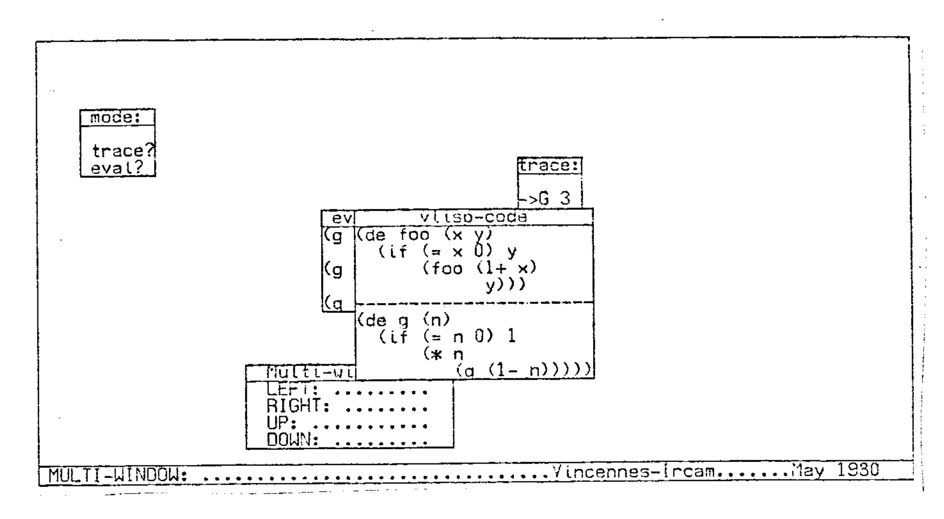
1. INTRODUCTION

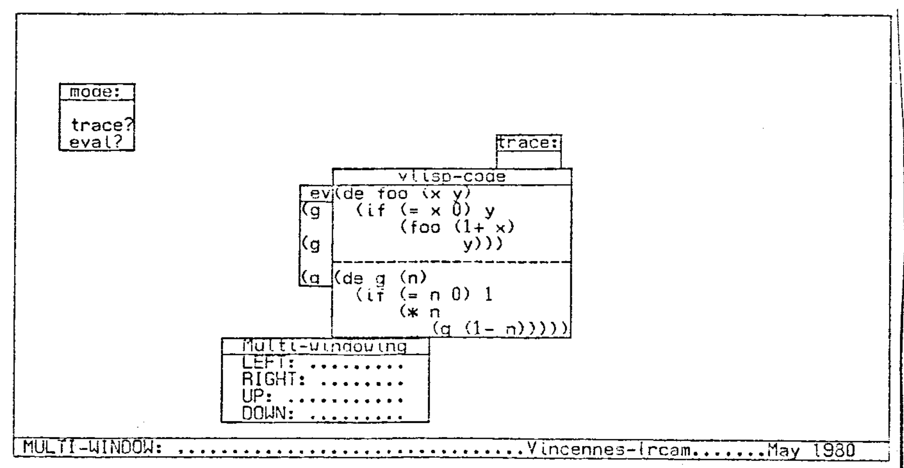
Nous avons construit, pour le système VLISP, un système de visualisation par fenêtres, déployables et déplaçables dynamiquement, tant par l'utilisateur en mode interactif que par programme, sous la forme d'appels de fonctions VLISP spécialisées.

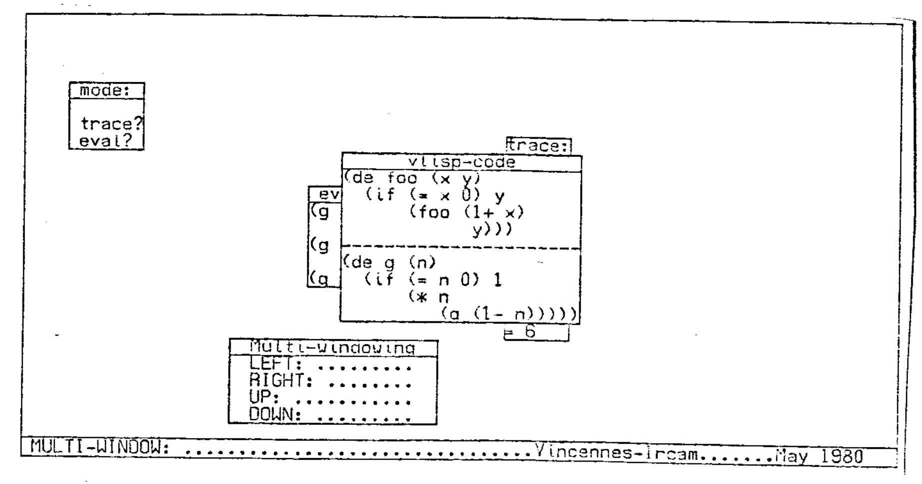
Ce système de fenêtrage dynamique, WIN, est illustré dans ce qui suit par un bref scénario.

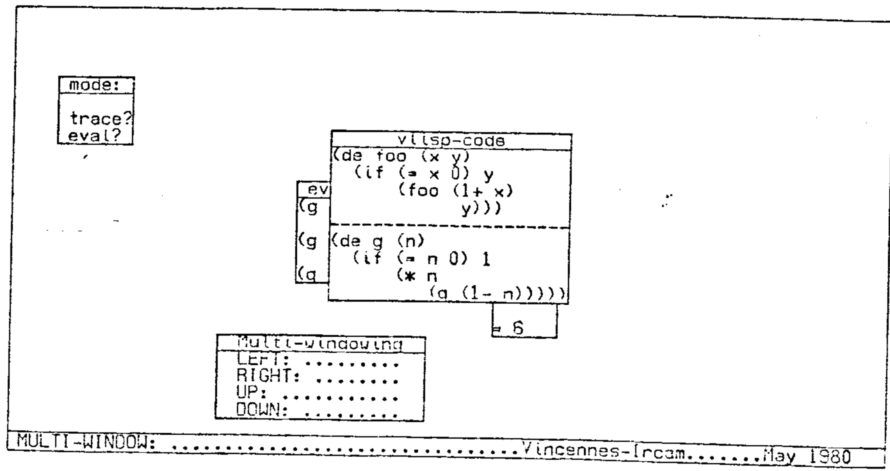


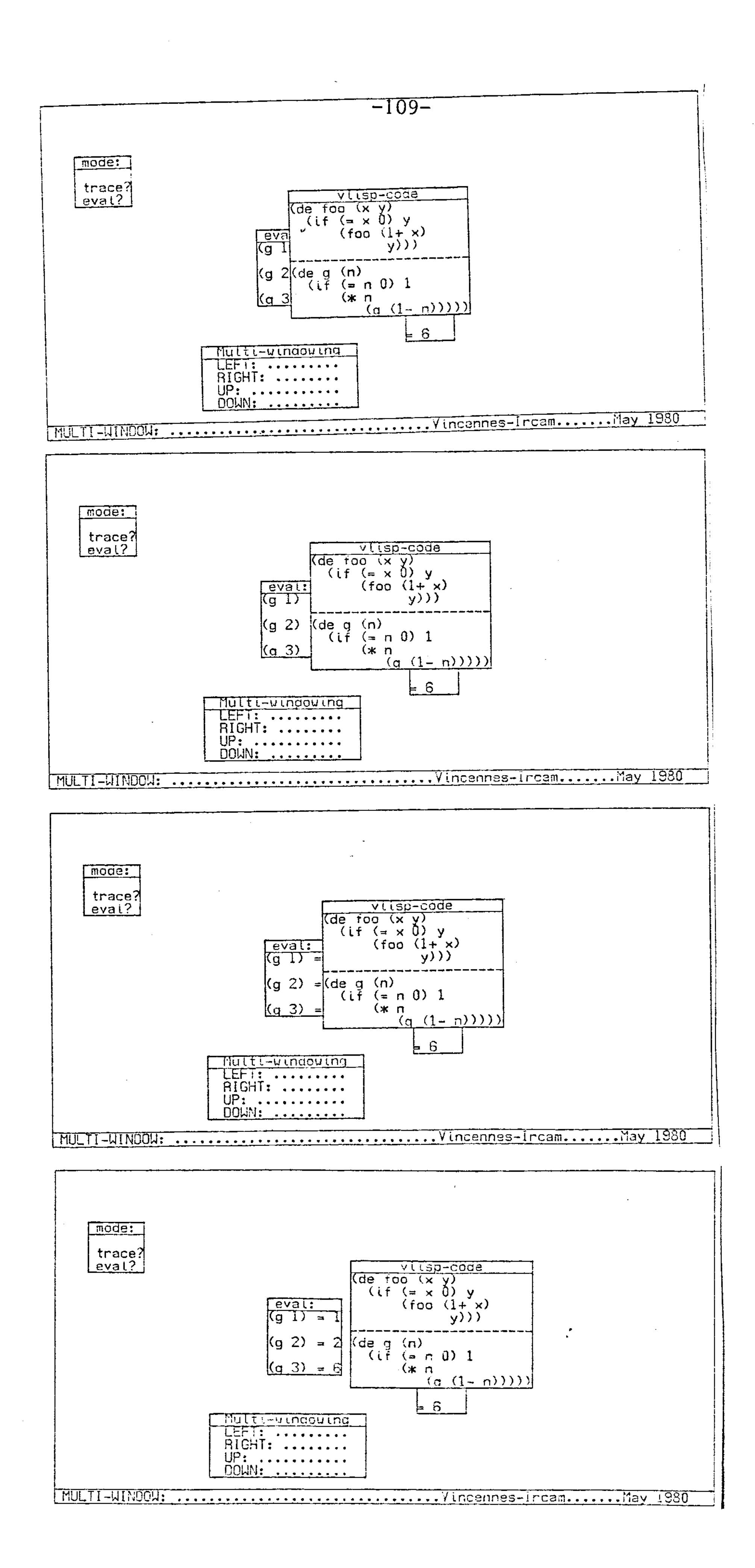



mode:
trace?
eval?
trace:
->G 3
ev
(g
(g
(g
vlisp-code
(de foo (x y)
(if (= x 0) y
(foo (1+ x)
y)))
(de g (n)
(if (= n 0) 1
(* n
(g (1- n)))))
Multi-wi
LEFT: ....
RIGHT: ....
UP: ....
DOWN: ....
MULTI-WINDOW: ....Vincennes-Ircam.......May 1980


mode:
trace?
eval?
trace:
vlisp-code
ev
(g
(g
(g
(de foo (x y)
(if (= x 0) y
(foo (1+ x)
y)))
(de g (n)
(if (= n 0) 1
(* n
(g (1- n)))))
Multi-windowing
LEFT: ....
RIGHT: ....
UP: ....
DOWN: ....
MULTI-WINDOW: ....Vincennes-Ircam.......May 1980

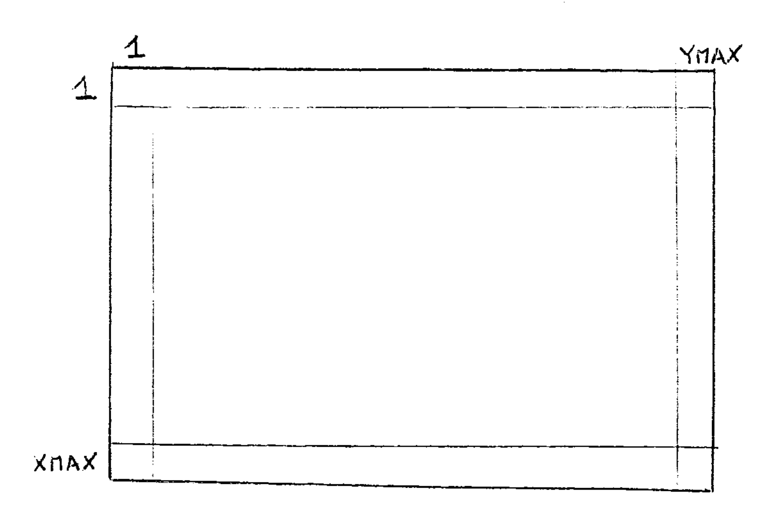

mode:
trace?
eval?
trace:
vlisp-code
(de foo (x y)
ev
(if (= x 0) y
(g
(foo (1+ x)
y)))
(g
(de g (n)
(g
(if (= n 0) 1
(* n
(g (1- n)))))
= 6
Multi-windowing
LEFT: ....
RIGHT: ....
UP: ....
DOWN: ....
MULTI-WINDOW: ....Vincennes-Ircam.......May 1980


mode:
trace?
eval?
vlisp-code
(de foo (x y)
(if (= x 0) y
ev
(foo (1+ x)
(g
y)))
(g
(de g (n)
(if (= n 0) 1
(g
(* n
(g (1- n)))))
= 6
Multi-windowing
LEFT: ....
RIGHT: ....
UP: ....
DOWN: ....
MULTI-WINDOW: ....Vincennes-Ircam.......May 1980


mode:
trace?
eval?
vlisp-code
(de foo (x y)
(if (= x 0) y
(foo (1+ x)
y)))
(de g (n)
(if (= n 0) 1
(* n
(g (1- n)))))
= 6
Multi-windowing
LEFT: ....
RIGHT: ....
UP: ....
DOWN: ....
MULTI-WINDOW: ....Vincennes-Ircam.......May 1980
eval:
(g 1)
(g 2)
(g 3)
(g 1) =
(g 2) =
(g 3) =
(g 1) = 1
(g 2) = 2
(g 3) = 6

2. DEFINITIONS

L'écran du terminal comporte XMAX lignes et YMAX colonnes.



La position courante du curseur est décrite par le couple de coordonnées: <**xc**, **yc**>, **xc** désignant le rang de la ligne courante, **yc** désignant le rang de la colonne courante.

La procédure:

tyo(*caractère*)

imprime le *caractère* sur la position courante du curseur, et avance le curseur d'une colonne. On notera que dans nos algorithmes, l'écran ne sera *pas* représenté globalement en mémoire.

Certains caractères, dits d'échappement, n'apparaitront pas visiblement, mais induiront une modification globale de l'écran, ainsi:

tyo(*home*)

rangera le curseur dans le coin supérieur gauche de l'écran, de même que:

tyo(*cleos*)

effacera l'écran à partir de la position courante du curseur.

La procédure:

poscur(**xc**, **yc**)

placera le curseur dans la position correspondant aux valeurs des variables **xc**, **yc**.

L'ensemble des fenêtres apparaissant sur l'écran sera organisé comme une liste à chaînage bi-directionnel représentant une queue de priorité, chaque élément de la queue étant constitué des champs suivants:

- **suiv** : adresse de la fenétre immédiatement plus prioritaire.
- **prec** : adresse de la fenêtre immédiatement moins prioritaire.
- **numf** : numéro de la fenêtre.
- **xf** : x-coordonnée coin supérieur gauche de la fenêtre.
- **yf** : y-coordonnée coin supérieur gauche de la fenêtre.
- **ndisjpp** : adresse de fenêtre plus prioritaire non-disjointe.
- **xcf** : x-coordonnée curseur privé de la fenêtre.
- **ycf** : y-coordonnés curseur privé de la fenêtre.
- **nl** : nombre de lignes de la fenêtre.
- **nc** : nombre de colonnes de la fenêtre.
- **contf** : adresse de la chaîne-contenu de la fenêtre.

3. QUEUE DE PRIORITE

La liste bi-directionnelle des fenêtres comporte une *fenêtre-butée* servant de tête de liste, d'adresse **premfen** qui ne comportera qu'un unique champ, le champ **suivant**. La fenêtre **premfen** sera toujours la *moins prioritaire*.

La derniére fenêtre de la liste sera toujours la fenêtre la plus récemment créée ou déplacée, son adresse sera repérée par la variable **fcour**. Le champ **suivant** de cette fenêtre contiendra le marqueur de fin de liste **eoq**.

Dans ce qui suit on dira qu'une fenêtre récente est *plus prioritaire* qu'une ancienne.

Pour modifier cet ordre si on déplace une fenêtre quelconque d'adresse **adf**, i.e. pour la placer en position de priorité maximum, on utilisera la procédure:

```
Remettre-en-Tête:
     si adf ≠ fcour alors
        suiv(prec(adf)) ← suiv(adf) ;
        prec(suiv(adf)) ← prec(adf) ;
        suiv(fcour) ← adf ; suiv(adf) ← eoq ;
        prec(adf) ← fcour ; fcour ← adf
     fsi
```

3. VISUALISATION DES CONTENUS DE FENETRES

Pour afficher isolément sur l'écran une fenêtre d'adresse **adf**, on utilisera la procédure suivante:

```
Afficher-Fenêtre:
     xc ← xf(adf) ; yc ← yf(adf) ; adr ← contf(adf) ;
     répéter nl(adf) fois
         poscur(xc, yc) ;
         répéter nc(adf) fois
             tyo(m[adr]) ; adr ← adr + 1
         frépéter
         xc ← xc + 1
     frépéter
```

Une premiére méthode, trés grossiére, d'affichage de *toutes* les fenêtres de la queue serait (accompagnant chaque déplacement *relatif* du curseur et de la fenêtre qui lui est temporairement attachée: **monter, descendre, avancer, reculer**):

```
Afficher-Toutes-les-Fenêtres:
     xc ← 1 ; yc ← 1 ; tyo(home) ; tyo(cleos) ;
     adf ← suiv(premfen) ;
     tant-que adf ≠ eoq faire
          Afficher-Fenêtre ;
          adf ← suiv (adf)
     ftant-que
```

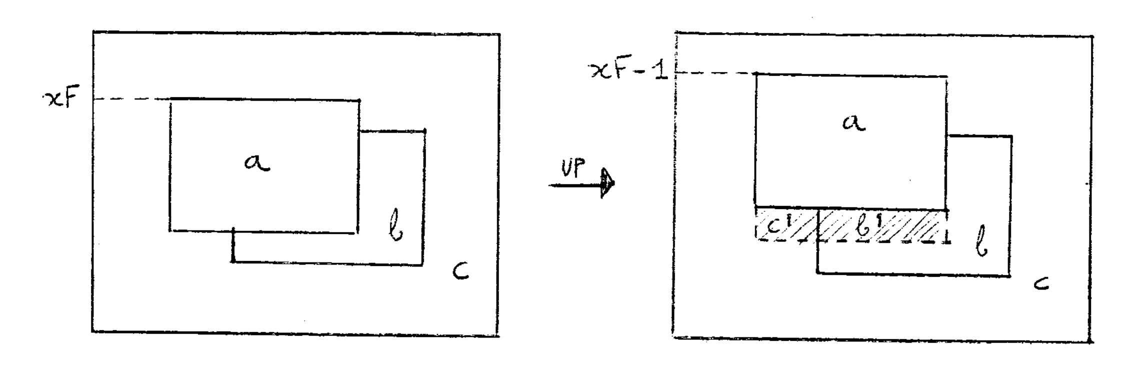
On notera qu'en cas de fenêtres non-disjointes, les caractères des fenêtres les plus récentes masqueront les caractères des fenêtres plus anciennes.

On obtient ainsi un premier procédé de multi-fenêtrage mobile.

4. AFFICHAGE MINIMAL

La méthode **Afficher-Toutes-les-Fenêtres** provoque des transitions écran-écran trop brutales, demeure trés inefficiente, en nécessitant la réécriture de ***tout*** l'écran à chaque transition, se prête enfin difficilement au tracé d'un *cadre* de fenêtre si la définition de l'écran en permet la visualisation.

Ré-examinons à nouveau le probléme du déplacement progressif d'une fenêtre. Prenons, à titre d'exemple, une configuration d'écran à 2 fenêtres non-disjointes:



La remontée de **A** doit provoquer l'apparition des zones:

B' appartenant à la fenêtre **B**

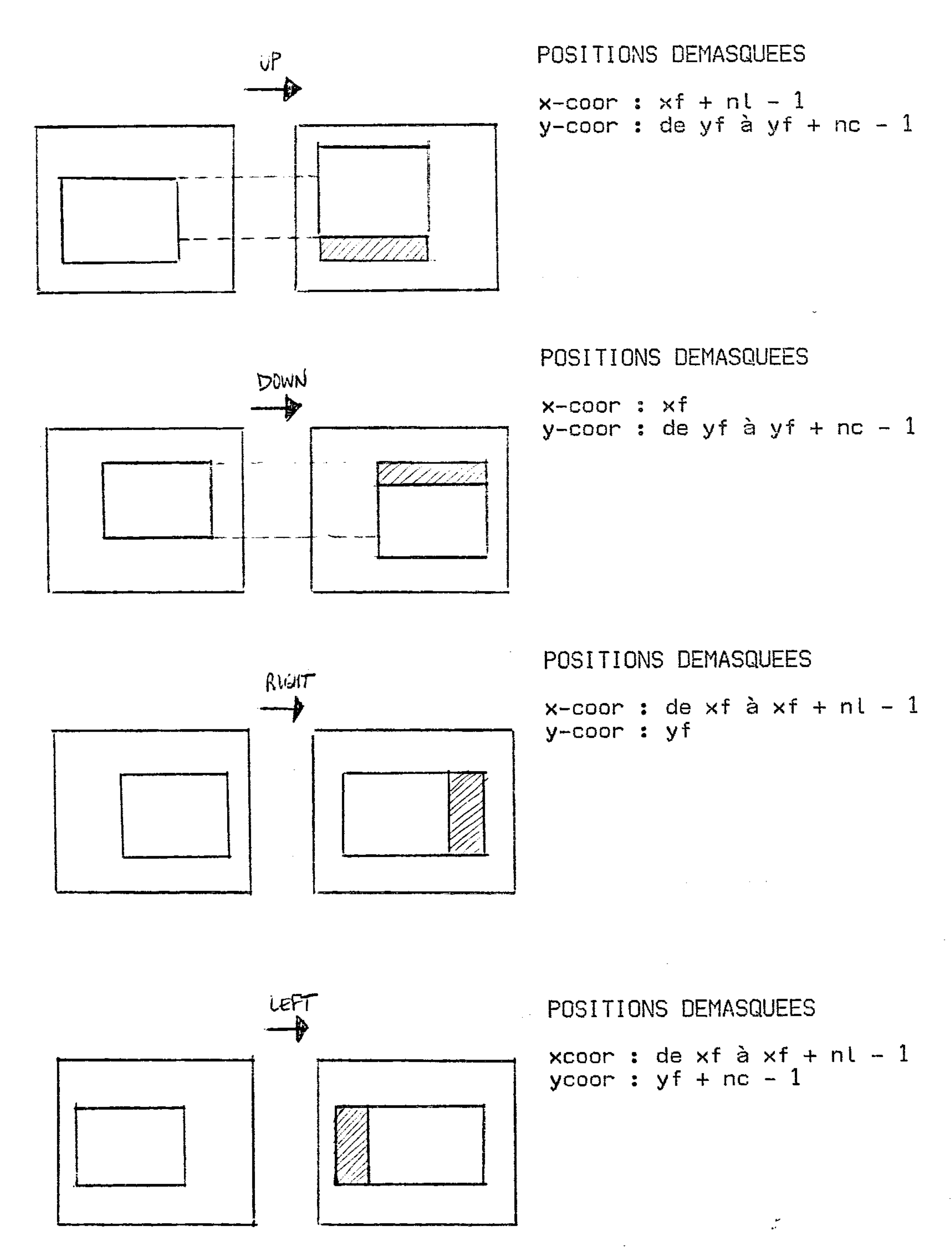
C' appartenant au fond **C**

Ceci suggére alors de traduire l'action demandée par la commande de curseur **haut** sous la forme de la séquence d'actions

1) réécrire **A** une ligne plus haut

2) pour chacune des positions de la zone pointillée libérée par le déplacement de **A**, rechercher quelle est la fenêtre **F** la plus prioritaire des fenêtres moins prioritaires que **A** à qui appartient la position (elle sera nécessairement ***visible***) et écrire le caractère correspondant de **F**. Si la position n'appartient à aucune fenêtre, écrire un espace.

En systématisant cette idée, on obtient alors les 4 cas suivants à considérer:



POSITIONS DEMASQUEES

x-coor : xf + nl - 1
y-coor : de yf à yf + nc - 1

POSITIONS DEMASQUEES

x-coor : xf
y-coor : de yf à yf + nc - 1

POSITIONS DEMASQUEES

x-coor : de xf à xf + nl - 1
y-coor : yf

POSITIONS DEMASQUEES

xcoor : de xf à xf + nl - 1
ycoor : yf + nc - 1

xf, yf, nl, nc étant les valeurs des champs correspondants de la fenêtre déplacée.

Nous obtenons, à la suite de cette analyse par cas, les algorithmes suivants:

Rechercher à quelle fenêtre *moins prioritaire* que **FCOUR** appartient la position de coordonnées <xc,yc> et ramener dans **adf** l'adresse de la fenêtre qui possède cette position. Si la position visible n'appartient à aucune fenêtre, ramener la valeur de **premfen** (la butée) dans **adf**.

```
Fenêtre-de-Position:
    adf ← prec(fcour) ;
    tant-que adf ≠ premfen faire
        si xc ∈ [xf(adf), xf(adf) + nl(adf) - 1] alors
        sinsi yc ∈ [yf(adf), yf(adf) + nc(adf) - 1] alors
        sinon exit
        fsi
        adf ← prec(adf)
    ftant-que
```

On balaye donc la liste des fenêtres, à partir de la précédente de **fcour**, en ordre de priorité décroissante.

Etant donné une fenêtre d'adresse **adf** qui posséde la position d'écran de coordonnées **<xc,yc>**, ramener dans **caractère** le caractère correspondant de la fenêtre (on transforme les coordonnées-écran absolues **<xc,yc>** en coordonnées relatives à la fenêtre). Si cette position n'appartient à aucune fenêtre, placer le caractère "espace" dans **caractère.**

```
Caractère-de-Position:
    si adf = premfen alors caractère ← " "
    sinon
        rxc ← xc - xf(adf); ryc ← yc ← yf(adf) ;
        caractère ← m[contf(adf) + nc(adf) * rxc + ryc]
    fsi
```

Imprimer le caractère correspondant à la position **<xc,yc>** de la plus prioritaire des fenêtres moins prioritaires que **fcour.**

```
Imprimer-Caractère:
    Fenêtre-de-Position ;
    Caractère-de-Position ;
    tyo(caractère) ;
```

On obtient enfin les algorithmes de mouvement ascendant, descendant, gauche et droite de la fenêtre courante, d'adresse **fcour**:

Monter:
```
    adf ← fcour ; xf(adf) ← xf(adf) - 1 ;
    Afficher-Fenêtre ;
    xc ← xf(adf) + nl(adf) ;
    yc ← yf(adf) ;
    poscur(xc, yc) ;
    répéter nc(adf) fois
      Imprimer-Caractère ;
      yc ← yc + 1
    frépéter
```

Descendre:
```
    adf ← fcour ; xf(adf) ← xf(adf) + 1 ;
    Afficher-Fenêtre ;
    xc ← xf(adf) - 1 ;
    yc ← yf(adf) ;
    poscur(xc, yc) ;
    répéter nc(adf) fois
      Imprimer-Caractère ;
      yc ← yc + 1
    frépéter
```

Avancer:
```
    adf ← fcour ; yf(adf) ← yf(adf) + 1 ;
    Afficher-Fenêtre ;
    yc ← yf(adf) - 1 ;
    xc ← xf(adf) ;
    répéter nl(adf) fois
      poscur(xc, yc) ;
      Imprimer-Caractère ;
      xc ← xc + 1
    frépéter
```

Reculer:
```
    adf ← fcour ; yf(adf) ← yf(adf) - 1 ;
    Afficher-Fenêtre ;
    yc ← yf(adf) + nc(adf) ;
    xc ← xf(adf) ;
    répéter nl(adf) fois
      poscur(xc, yc) ;
      Imprimer-Caractère ;
      xc ← xc + 1
    frépéter
```

Sur ces bases, de nombreuses améliorations sont possibles, essentiellement dépendantes de la place mémoire disponible, parmi lesquelles:

- introduire un ou plusieurs champs supplémentaires dans les fenêtres permettant de trier les fenétres par ordre de **xf**, **yf**, **xf+nl**, **yf+nc** croissants, pour accélérer la recherche.

- si les caractères sont stockés sur 7 bits, utiliser le 8éme bit comme bit de marquage, placé par la fenêtre à qui *appartient* la position absolue correspondant à ce caractère.

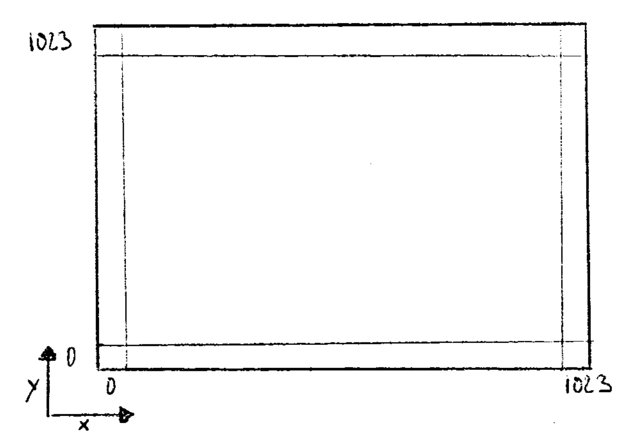
- réorganiser le système sous forme d'un système de *passage de messages*: les couples <fenêtre, position> seront considérés comme des acteurs recevants et s'envoyant mutuellement des messages d'occupation et de libération de positions.

5. BORDS

Le tracé efficace de *bords* de fenêtres, sous forme de **vecteurs**, qui devront être également déplaçables et masquables, sera relativement plus délicat. Nous examinerons une série de solutions d'efficacité croissante et de simplicité décroissante.

Naturellement ces bords, pour être mis en jeu, nécessitent un écran convenable: de type display-vecteur ou bit-map. Les algorithmes qui suivent concerneront un écran de type display-vecteur, tel que le VT11 de la compagnie DEC.

L'écran est encore une matrice de positions adressables en coordonnées x-y, que nous supposerons disposer de 1024 lignes et 1024 colonnes.



Le contrôleur de cet écran doit être conçu comme un processeur séparé à part entiére, mettant en jeu les instructions (d-v-instructions) suivantes:

{POINT x y}

Se positionner sur l'écran à la position <x, y>.

{LONGV Δx Δy}

Tracer sur l'écran, à partir de la position courante, un vecteur de longueur définie par l'incrément horizontal Δx et par l'incrément vertical Δy.

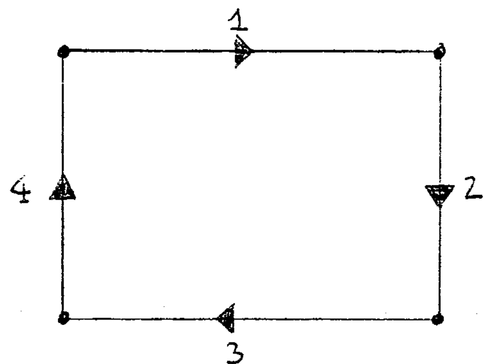
Un bit positionné dans l'incrément Δ décidera si le tracé de vecteur a lieu dans la direction horizontale décroissante: **minusx** ou dans la direction horizontale décroissante: **minusy**.

{DRET}

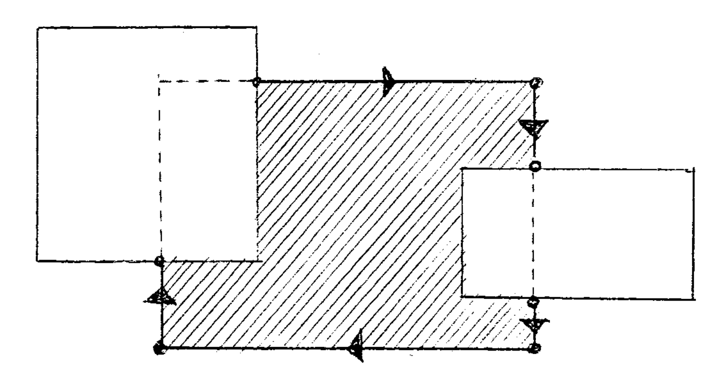
Re-exécuter tout le d-v-programme à partir de sa premiére instruction. On nommera ***rafraîchissement*** de l'écran une re-exécution du d-v-programme.

Comment afficher les bords d'une fenêtre de la liste des fenêtres ?

Si la fenétre était *totalement* visible, nous aurions à tracer les 4 vecteurs orientés orthogonaux



Si, en revanche, la fenétre n'était que partiellement visible, ce serait l'effet de fenêtres plus prioritaires partageant des positions de sa surface. Nous pourrions, à titre d'exemple, avoir les 5 vecteurs suivants:



Une premiére procédure de génération du d-v-programme correspondant sera alors:

> Pour chacun des bords 1, 2, 3 et 4, construire à partir de la premiére position visible <**xp**, **yp**>, un vecteur de longueur égale au nombre de positions visibles à partir de celle-ci dans cette direction, et recommencer à partir de la prochaine position visible du bord.

Comment déterminer, pour une fenétre **adf**, si une position de coordonnées <**xR**, **yR**> est ou non visible ? Cette position sera effectivement visible si elle n'appartient pas à une fenêtre plus prioritaire que **adf**.

On a alors l'algorithme:

```
VISIBLE:
    auxADF ← suiv(adf) ;
    tant-que auxADF ≠ 0 faire
      si    xR ∉ [xf(auxADF), xf(auxADF) + nl(auxADF) - 1] alors
      sinsi yR ∉ [yf(auxADF), yf(auxADF) + nc(auxADF) - 1] alors
      sinon visible ← faux ; exit
      fsi
      auxADF ← suiv(auxADF)
    ftant-que
    visible ← vrai
```

Comment obtenir, dans xP et yP, les coordonnées physiques de l'écran-d-v correspondant aux coordonnées <xf, yf> de la position supérieure gauche d'une fenêtre d'adresse **adf** ?

L'algorithme suivant mettra en jeu les constantes suivantes:

homex:	x-coordonnée de la position supérieure gauche du d-v-écran.
homey:	y-coordonnée de la position supérieure gauche du d-v-écran.
car.largeur:	largeur d'un caractère.
car.hauteur:	hauteur d'un caractère.

```
POSITION-BORD-FENETRE:
    xp ← homex + car.largeur * (yf(adf) - 1) ;
    yp ← homey + car.hauteur * xf(adf)
```

On obtiendrait, à titre d'exemple, pour le bord 1 de la fenêtre **adf**, l'algorithme suivant, utilisant le booléen **programme-généré**: ce dernier sera *vrai* lorsque la longueur ∆v d'un vecteur est en cours de construction pour un segment visible du bord. La procédure **Construire** place ses arguments à la suite des d-v-instructions du d-v-programme en construction.

```
Position-Bord-Fenêtre ;
xR ← xf(adf) ; yR ← yf(adf) ;
programme-généré ← faux ;
répéter nc(adf) fois
  si visible alors
     si programme-généré alors
       ∆v ← ∆v + car.largeur
     sinon
       Construire{POINT xp yp} ;
       ∆v ← car.largeur ;
       programme-généré ← vrai
     fsi
  sinsi programme-généré alors
     Construire{LONGV ∆v 0} ;
     programme-généré ← faux
  fsi
  xP ← xP + car.largeur ;
  yR ← yR + 1
frépéter
si programme-généré alors
   Construire{LONGV ∆v 0} ;
   programme-généré ← faux
fsi
yR ← yR - 1
```

La généralisation de cet algorithme nous donne alors l'algorithme d'affichage de bord d'une fenêtre d'adresse adf:

```
BORD-FENETRE:
   Position-Bord-Fenêtre ;
   xR ← xf(adf) ; yR ← yf(adf) ;
   programme-généré ← faux ;
   Bord(nc(adf), car.largeur, car.largeur, xP, Δv, 0, yR, 1) ;
   Bord(nl(adf), car.hauteur, -car.hauteur, yP, 0, Δv OR minusy, xR, 1) ;
   Bord(nc(adf), car.largeur, -car.largeur, xP, Δv OR minusx, 0, yR, -1) ;
   Bord(nl(adf), car.hauteur, car.hauteur, yP, 0, Δv, xR, -1)
```

qui utilisera la macro:

```
BORD(nbr, iΔv, iBordPos, BordPos, xΔv, yΔv, Pos, iPos)
   répéter nbr fois
     si visible alors
        si programme-généré alors
          Δv ← Δv + iΔv
        sinon
          Construire{POINT xP yP} ;
          Δv ← iΔv ; programme-généré ← vrai
        fsi
     sinsi programme-généré alors
        Construire{LONGV xΔv yΔv} ;
        programme-généré ← faux
     fsi
     BordPos ← BordPos + iBordPos ;
     Pos ← Pos + iPos
   si programme-généré alors
      Construire{LONGV xΔv yΔv} ;
      programme-généré ← faux
   fsi
   Pos ← Pos - iPos
```

L'algorithme d'affichage de bords de *toutes* les fenêtres de la liste des fenêtres devient alors:

```
"stopper le d-v-programme d'affichage de bords" ;
adf ← fcour ;
tant-que adf ≠ premfen faire
   Bord-Fenêtre ;
   adf ← prec(adf)
ftant-que
Construire{DRET} ;
"redémarrer le d-v-programme d'affichage de bords"
```

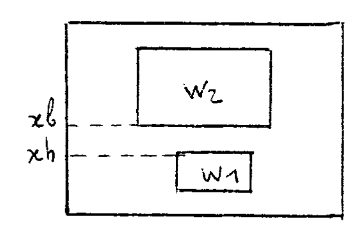
Cet algorithme, quoique correct, demeure relativement inefficient: une premiére amélioration peut consister à afficher, au fur et à mesure de leur construction, les {POINT x y} et les {LONGV ∆x ∆y}, afin d'obtenir un réaffichage plus continu.

Par ailleurs, il n'est pas nécessaire de recalculer, pour ***toutes*** les positions de bord d'une fenêtre, si elles sont visibles dans cette fenêtre ou masquées par une fenêtre plus prioritaire.

Une premiére amélioration consiste à déterminer en temps constant si la fenêtre w1 dont on veut afficher le bord et une autre fenêtre w2 plus prioritaire sont ou non disjointes. Si elles sont effectivement disjointes, cette fenêtre disjointe prioritaire pourra étre *ignorée*, pour toutes les positions de bord de la fenêtre candidate à l'affichage, dans le test de visibilité. Ce pré-test améliorera alors considérablement la vitesse de fabrication du d-v-programme d'affichage.

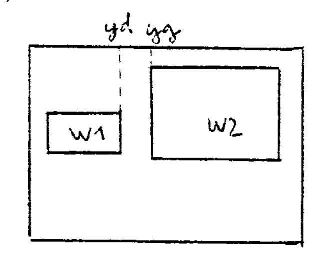
On peut distinguer, en premiére approche, 4 cas de conditions suffisantes de disjonction.

CAS 1



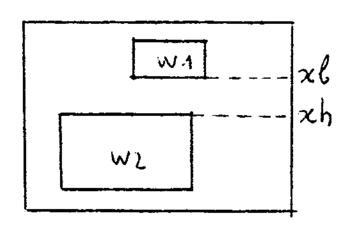
xh(w1) > xb(w2)

CAS 2



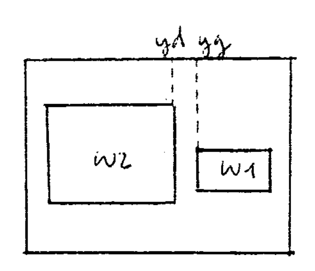
yd(w1) < yg(w2)

CAS 3



xb(w1) < xh(w2)

CAS 4



yg(w1) > yd(w2)

avec xh = xf , xb = xf + nl - 1
yg = yf , yd = yf + nc - 1

Le pré-test de visibilité consistera à constituer, à partir de la liste des fenêtres plus prioritaires, la sous-liste des fenêtres *non*-disjointes. L'algorithme de visibilité d'une position ne testera le masquage possible de cette position ***que*** pour ces fenêtres non-disjointes.

Dans le cas où une fenêtre d'adresse ADF, de priorité quelconque, est disjointe de toutes les autres (liste des fenêtres non-disjointes vide), ce qui s'applique également à la fenêtre courante FCOUR, l'algorithme direct d'affichage total des bords sera mis en jeu dans la fabrication du d-v-programme:

```
Construire{POINT xp yp} ;
xD ← car.largeur * nc(adf) ; yD ← car.hauteur * nl(adf) ;
Construire{LONGV xD 0} ;
Construire{LONGV 0 yD CR minusy} ;
Construire{LONGV xD CR minusx 0} ;
Construire{LONGV 0 yD} ;
```

On notera qu'après l'execution d'un LONGV, la position courante sur le d-v-écran est celle de l'extrêmité *finale* du vecteur affiché par LONGV.

6. REFERENCES

[1] CHAILLOUX J. *Le modèle VLISP: description, implémentation et évaluation*, (Thése de 3e cycle), Rapport LITP no 80-20, Avril 1980

[2] DEC *VT11 graphic display processor*, EK-VT11-TM-001, Digital Equipment Corporation, Maynard Mass., September 1974

[3] GOLDBERG A., KAY A. (eds) *SMALLTALK-72 Instruction Manual*, SSL76-6, Xerox PARC, Palo Alto, Ca., March 1976

[4] GREUSSAY P. *VLISP-11 Reference Manual*, Université Paris-8-Vincennes, (à paraître), 1980

[5] NEWMAN W.M., SPROULL R.F. *Principles of Interactive Computer Graphics*, 2d edition, Mc Graw Hill, 1979

[6] RIEGER C. *The LISP Window Slave*, Unpublished Manuscript, M.I.T. Artificial Intelligence Laboratory, November 1975

[7] SPROULL R.F. *The design of gray-scale graphics software*, Proc. of the IEEE Conf. on Computer Graphics, Pattern Recognition and Data Structure, May 1975, 18-20

[8] SPROULL R.F. *Raster Graphics for Interactive Programming Environments*, CSL-79-6, Xerox PARC, Palo Alto, Ca., June 1979

[9] TEITELMAN W. *A Display-Oriented Programmer's Assistant*, Proc. 5th Int. Joint Conf. Artificial Intelligence, August 1977, 905-915

[10] WARREN S.K., ABBE D. *Rosetta Smalltalk: A conversational, Extensible Microcomputer Language*, SIGSMALL Newsletter, Vol 5, no 2, April 1979, 13-22

[11] WEINREB D., MOON D. *Lisp Machine Manual*, 2nd preliminary version, M.I.T. AI Lab, January 1979